Communication in Microkernel-Based Operating Systems

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Diplom-Informatiker Ronald Aigner
geboren am 20. Mai 1976 in Dresden

Gutachter:
Prof. Dr. rer. nat. Hermann Härtig — Technische Universität Dresden
Prof. Dr. Klaus Meyer-Wegener — Friedrich Alexander Universität Erlangen-Nürnberg

Tag der Verteidigung: 21. Januar 2011

Dresden im Mai 2011

Communication in Microkernel-Based Operating Systems

Ronald Aigner
Technische Universität Dresden

July 2010

Abstract

Communication in microkernel-based systems is much more frequent than system calls known from monolithic kernels. This can be attributed to the placement of system services into their own protection domains. Communication has to be fast to avoid unnecessary overhead. Also, communication channels in microkernel-based systems are used for more than just remote procedure calls. In distributed systems, which also have a componentized design, it is state of the art to use tools to generate stubs for the communication between components. The communication interfaces of components are described in an interface definition language (IDL). In contrast to distributed systems, components of a microkernel-based system run on the same architecture and message delivery is guaranteed.

In this Thesis, I explore the different kinds of communication, which can be used in microkernel-based systems, as well as their possible representation in IDL. Specifically, I introduce the syntax to describe kernel objects in IDL. I discuss the complexity of IDL compilers and its relation to the complexity of the IDL. Furthermore, I evaluate the performance of the communication stubs generated by different IDL compilers and discuss techniques to minimize performance overhead in generated stubs. I validated these techniques by implementing the Drops IDL Compiler – Dice. Finally, this Thesis presents a mechanism to measure the frequency and performance of invocations of generated communication code. I used this technique to conduct measurements in highly complex systems and introducing the least possible overhead.

Acknowledgments

First, I would like to thank my supervisor, Prof. Hermann Härtig. He provided the environment to pursue my research ideas and to learn so much about operating systems and microkernels in particular. I want to express my gratitude for not giving up on me and supporting this work. He created a nourishing environment by shepherding so many intelligent people in his group.

Furthermore, I would like to thank my reviewer, Prof. Meyer-Wegener. His infinite patience and positive feedback allowed me to look at this work from different angles and find new ideas behind the corners.

Of course this Thesis would not have been possible without all the wonderful people of the Operating Systems chair at TU Dresden. Discussions with them always questioned my assumptions and made me constantly aware of flaws in my work. I would especially like to thank Lars Reuther, Martin Pohlack and Adam Lackorszynski.

Last but certainly not least, I wish to thank my family. My wife and kids gave me the balance to stay on track, even if the work seemed overwhelming. And my parents: You made me who I am.

Contents

List of Tables

Chapter 1

Introduction

Communication in microkernel-based systems has always been the focus of research in the operating systems community. The performance of the interaction between various components of an operating system at all times influences the performance of the overall system. In this Thesis I will analyze the various kinds of interaction between components of the Dresden Real-time OPerating System (DROPS) as an example of microkernel-based systems. I will isolate the influence of compilers, communication code and the hardware on which the code runs. And I will describe a method how tools, namely stub code generators, can help to use efficient communication code for the interaction of components of a microkernel-based operating system.

An operating system provides its functionality by a set of *services*. These services are often implemented in distinct *components*, which communicate via *interfaces*. Microkernel-based operating systems emphasize the separation by placing distinct components into respective protection domains. Because communication between components crosses protection domain boundaries, microkernel-based systems have similarities with *distributed object computing*, where communication with a service crosses nodes.

The code to invoke a procedure of a "remote" component – to make a *remote procedure call* – is divided into *client* and *server* part and follows the same pattern for all services: The client communication code first packs the parameters of an invocation into the *message* (i.e., *marshals*), sends the message to the server and waits for a reply. The server communication code waits for a request, unpacks input parameters, i.e., *unmarshals*, dispatches the request, marshals output parameters, and sends the reply. The client code then unmarshals the return values from the reply.

For distributed systems it is state of the art to use tools to generate this communication code. The description of a server interface, written in an *interface description language* (IDL), is translated into communication code – also called stubs – by an *IDL compiler* or *stub code generator*. Most microkernel-based systems also use IDL compilers to generate communication stubs, for instance MIG for Mach, IDL[4] for Pistachio, or DICE for L4.Fiasco.

The usage of tools to generate communication code reduces common software errors in the communication code, such as typos or mismatches between client side and server

side parameters. An interface description is much simpler to read and errors much easier to find than in hand-written communication code. Furthermore, using a tool allows migrating a component to a different communication platform easily by adapting only one piece of software: the IDL compiler.

IDL compilers always generate code for a specific communication platform. For distributed systems, such a platform is usually the operating system's socket interface. For microkernel-based systems such a target platform is the L4 microkernel. Because the *inter-process communication* (IPC) of L4 microkernels is fast, the share of computation time for marshaling and unmarshalling data is, compared to the communication itself, important. Therefore, one requirement for IDL compilers for microkernels is that the generated communication code requires minimum time.

As mentioned before, *remote procedure calls* (RPCs) are used to communicate between components and to invoke services. Additionally to RPC, microkernel-based systems use communication to signal state between threads with simple messages. Or applications stream large amounts of data using shared memory.

Other restrictions that apply to communication in microkernel-based systems include: Lower level services cannot use dynamic memory management (heaps) or a thread library. An IDL compiler for microkernel-based systems has to generate code that works without this functionality. Also, developers of low level component have to have fine-grained control over resource usage in the communication code. Heaps or unbounded stacks are often unavailable. Multi-threaded clients and servers should be able to use the generated code without provoking a deadlock.

As mentioned above, the usage of an IDL compiler allows to easily retarget software to different communication platforms. An alternative approach is to use a communication library that can be used on different communication platforms. But, besides allowing a maximum of portability, we also want to achieve a maximum of performance, which requires the exploitation of all capabilities of the underlying communication mechanism. Using a generic communication library does not allow to exploit all features of the microkernel's communication mechanisms.

On L4 microkernel-based systems transferred data can also have additional semantic. Flexpages, for instance, specify a memory range that should be made accessible in the address space of the communication target. Most recent L4 versions generalize this feature to transfer capabilities, i.e., generic rights to objects, to another thread. Other special semantics include indirect data transfer, where the message only contains base address and size of a memory region and the kernel copies the data directly from the source address to the target address.

Besides the previously mentioned requirements on an IDL compiler, there also exist requirements, which, on first sight, do not belong to communication.

One such requirement is the real-time property of a server. A real-time capable server should communicate using real-time capable stubs, that is, they process messages in bounded time.

Another orthogonal requirement on communication is introduced by security. Monitors are one technique to perform access control for servers by restricting or filtering

requests. Such security measures could be automatically supported in the generated code.

Resource management in the communication code is important for real-time and security properties. If resource access in the communication code is not bounded, the response time of the entire service cannot be guaranteed. Also, denial of service attacks can exhaust resources required in the communication code, thus preventing the service to receive further requests.

However, adding a new feature or kind of communication to an IDL compiler has to be thoroughly discussed. The costs of a feature, such as code complexity or performance degradation of the generated code, have to be weighed against the benefit for the user. It might be possible to implement a feature as a wrapper, which avoids the additional costs for the generic case, but allows exploiting a specific feature for a special case.

This Thesis will contribute to the state of the art by discussing the following three parts:

- In microkernel-based systems different communication forms are used. Supporting these different forms of communication in a tool that generates communication code is essential. This Thesis includes a detailed analysis of the most common forms of communication in microkernel-based systems and will introduce means to describe these forms of communication in an interface description language. Cross-cutting problems, such as resource management or security will be discussed as well.

- A microkernel API changes more frequently than the API of a monolithic kernel. Software running on top of the microkernel has to adapt to these changes and an IDL compiler generating the code, which interfaces with the microkernel, can provide a level of abstraction. With a new microkernel API new potentials for optimization or new features are introduced. An IDL compiler will have to exploit these features, otherwise it becomes inefficient. This Thesis discusses a technique to exploit specific features of a microkernel API but still provide an interface description that is portable.

- Efficient tools are easy to learn and use. To minimize the learning curve, features known to the users have to be reused in a new tool with the same semantics or a minimum of differences. This Thesis shows how known techniques and features have been integrated into an IDL compiler.

In the rest of the Thesis I will first summarize related work. I will then discuss the claims of this Thesis in more detail. Chapter 6 will then evaluate my claims and finally I will conclude by giving an outlook on future work. As a practical instrument to validate and test the communication forms and features I implemented the Drops IDL Compiler (DICE) which is referenced throughout this Thesis.

Chapter 2

Related Work

The following chapter introduces and discusses work related to communication in microkernel-based systems. I will discuss communication mechanisms, such as Message Passing Interface, followed by different approaches to stub code generation. More detailed discussion of stub code generators for microkernel-based systems is included as well as a presentation of alternative approaches to specify interface contracts.

2.1 Message Passing Interface

The Message Passing Interface (MPI) [73, 24] was designed as means to ease communication between programs running on heterogeneous nodes. Programs connected with MPI implement massive parallel applications, such as matrix calculations. MPI is designed as a library, without the need for pre-processing or compilation. Thus, assumption about exchanged data cannot be made in advance, but the interface is generic enough to allow all sorts of data. Because the applications run in a heterogeneous environment, data has to be typed so necessary conversions can be detected and performed.

The MPI standard defines functions for sending and receiving messages, where a send or receive function takes as payload argument only a pointer to a buffer. To exchange multiple different parameters, either multiple sends are necessary or the parameters have to be packed into a byte array by wrapper functions. Due to the intended use for massive parallel computations, the MPI library only contains send and receive functions: A coordinator sends input data to the distributed computation programs and later collects the output from these programs. The caller was not suspended after sending the request until the reply was delivered. There was no intention to support remote procedure calls.

2.2 Remote Procedure Calls

To extend the availability of remote processing power to a broader range of services, the idea of remote procedure calls was widely discussed. Birrell and Nelson discuss one of the first implementations of RPC in [16]. They extend earlier ideas [81] with precise

semantics of behavior in case of communication failure, semantics of address-containing arguments in the absence of shared memory, how services are identified, and protocols for data and control transfer between caller and callee.

RPC is synchronous, that is, the caller is suspended when sending the request and only resumes after the reply has been received. The concept of RPC obsoleted communication channel setup as specified in the OSI network layers. Session-based communication can be implemented on top of RPC. Also, RPC is strictly peer to peer oriented whereas MPI is targeted to many to many communication.

2.3 Sun RPC

The Sun remote procedure call (RPC) [75]–or transport-independent remote procedure call (T1-RPC)–was developed by Sun and AT&T as part of the UNIX System V Release 4. It makes RPC applications transport-independent by allowing a single binary version of a distributed program to run on multiple transports, i.e., communication protocols. Previously, with transport-specific RPC, the transport was bound at compile time so applications could not use other transports unless the program was rebuilt. With TI-RPC, applications can use new transports if the system administrator updates the network configuration file and restarts the program. Thus, no changes are required to the binary application.

TI-RPC allows only a single parameter to be passed from client to server. If more than one parameter is required, the parameters have to be combined into a structured parameter. Reply information passed from server to client is passed as the function's return value.

Because T1-RPC was explicitly designed for network communication it deals with issues such as message loss, data representation, and interchangeable communication protocols. To identify a service, a tuple of *program number, version number, procedure number* is specified. The tool used to generate communication stubs for T1-RPC is `rpcgen`, one of the first tools to generate communication stubs from an IDL. Extended features of `rpcgen` include C-style argument passing to stubs (instead of a pointer to a structure containing the parameters), multithreading-safe stubs, and timeout handling for client and server.

2.4 Firefly

In [14] Bershad et al. describe optimizations for remote procedure calls (RPC) on local nodes, which are *copy avoidance* and *lazy scheduling*. These optimizations are achieved using four techniques described in the paper as: simple control transfer (lazy scheduling), simple data transfer (direct transfer of argument stack and thus copy avoidance), simple stubs, and design for concurrency (avoid shared data structure bottlenecks). The paper contains a detailed analysis of frequency of local and remote calls and the typical size of an RPC. The results of the analysis show that in a common machine setup less than 6% of all calls are to remote machines. To closer differentiate the local calls 28 RPC

services with 366 procedures have been instrumented and the numbers show that 95% of the RPC invocations have gone to 10 procedures and 75% to only 3 procedures. In the 95 percentile most calls were 50 bytes in size or less and the majority was less than 200 bytes in size. This paper describes the design of RPC services of a distributed system where the programmers had the distribution in mind.

Schroeder and Burrows describe in [86] some basic rules for RPC. They argue that only fast RPC will make programmers use it and to make RPC fast you have to be aware of what happens where and when. This is a very strong point for fine-grained analysis of generated code and usage patterns. Once the slow parts are identified they have to be rewritten. A major point made by Schroeder and Burrows is that message buffers should be used *in place* to avoid copies.

2.5 Spring

Spring [79] is a distributed system with about 100 modules, 200 interfaces, and 500 operations. Kessler describes in [63] approaches to use interpretive stubs instead of compiled stubs for RPC communication. A compiled stub is fixed for each method specified in the interface description and optimized for that method. If you have 500 operations in a system, the size of the stubs has a major impact on the overall size of the system. With a typical size of 300 bytes, as described in the paper, this is quite significant.

Kessler suggests using one interpretive stub that takes a condensed interface description for each method and translates that description together with the arguments of the call into the RPC. The interpreter described in the paper had a size of four kilobytes and an average description was only 27 bytes in size. In the paper no significant performance overhead was observed, but the measurements were made on a SPARC with a suboptimal C++ compiler. A benefit of this approach is the locality of the interpreter code: it can be used by every client via shared memory.

2.6 Pebble

The Pebble approach to inter-process communication integrates the stub code generation into the run-time system of operating system. In Pebble, which is a microkernel with protection domains, communication is done through *portals*, where portals are addressed via local names, i.e., indices into a process local portal table. A portal has invoker-side and invoked-side generated portal code that is protected from the user and generated by the portal manager and pasted into the address space of the application on portal creation. When creating a new portal the receiver specifies the possible parameters for the portal. To keep the size of the stub code generator small, communication only uses a small number of fixed sized parameters or memory pages.

The specification consists of three parts. The first one defines what to do with the stack on message transfer: either allocate a new stack at receiver's side or use the caller's stack. The second part specifies what to do with the caller's registers: protect them or

only save a minimal subset. And the last one contains the parameter specification, where a parameter can be a constant, an integer value, or a memory page (*window*). These elements are specified as characters in the portal specification. For example: "smcii" – share stack ('s'), minimal register protection ('m'), one constant and two integer parameters ('cii').

This approach is not viable for our microkernel-based system. Interpretive stubs always add performance overhead. In our system performance is paramount and used as the main factor to define stubs as acceptable or not.

2.7 MIG - The Mach Interface Generator

Mach [2] as a first generation microkernel uses a stub code generator – The Mach Interface Generator (MIG) [27] – to generate communication code. To understand the features of MIG I will first introduce some of Mach's basic concepts and mechanisms [60].

Mach uses *tasks* as execution environments in which threads may run. A task is the basic unit for resource allocation. A *thread* it the basic unit of CPU utilization. Threads communicate with each other using *messages*, which consist of data objects, which can be of any size, contain pointers and typed capabilities. The target of a message is a *port*. A port is also used as protected capability for all objects. The creating thread has ownership and receive rights on that port. In [94] Walmer and Thompson describe a *port set* as a collection of ports from which one thread may receive. Ports can also be part of a message and thus rights to invoke methods on objects can be transferred between threads. If no port set is used, one thread may only receive from one port.

The virtual memory (address space) of a task is managed by the Mach kernel. It implements the paging and replacement strategy for all tasks. A task may request additional chunks of virtual memory using system calls. Such chunks can be assigned to *memory objects* [97], which is a named resource that can be shared with other tasks. The creator of a memory object is its pager. The Mach kernel may create memory objects which are backed by a default pager.

Mach was designed as a distributed operating system on heterogeneous nodes in a network. To unify communication on local nodes and between different nodes all messages are in a platform independent format. The payload of a message is typed, i.e., it contains type identification for each member and the actual data. The receiver checks the type identification against its platform and has to convert data if required.

The payload may consist of *simple data*, *out-of-line data*, and *ports*. Simple data is stored in the message itself. Out-of-line data points to a region in virtual memory. At receiver's side memory is allocated using virtual memory calls of kernel and data is copied from sender to receiver. Because the Mach kernel does all the virtual memory handling, it will also guarantee that out-of-line data can be transferred across nodes.

2.7.1 Specification Language

The Mach Interface Generator (MIG) [27] is derived from Matchmaker and uses features of Matchmaker's specification language. MIG's specification language (MIG IDL) is similar to Pascal and allows the usage of simple types, structured types, arrays, and pointers. It also has the notion of a polymorphic type, which is similar to a union type in C/C++. It may be instantiated to any other IDL type at run-time. One special feature of MIG IDL is the support of types that change during transmission, which is implemented using *type translators*. Type translators are hooks defined in the interface specification, which are called at the server side after receiving a message and before a reply. The type is translated from the transport format into the server internal format and, before the reply, converted back into the transport format. Generally speaking one type has three representations: in the user (client) code, in the server stub, and in the server procedure.

MIG IDL allows the specification of different types of RPC: operations without return value (*procedure*), operations with a type as return value (*function*), and operations with error codes as return value (*routine*). In case of an error *procedures* and *functions* invoke callbacks instead of returning the error code as return type. *Procedures* and *routines* may also be send only for asynchronous invocations.

The definition of the parameters of all operations consists of type, variable name, an optional deallocation flag, and a specification, which determines the usage of that parameter. This can be either `in`, `out`, or `inout` indicating the direction of the transmission. Alternatively one may specify `RequestPort` or `ReplyPort` indicating the ports to send to or receive the reply from. Or a parameter can have the specification `WaitTime` indicating a timeout for the RPC. The specification may also indicate the message type, which can be *normal* for two message invocations: send and receive, *RPC* for one message invocation combining the send and receive, or *encrypted* for encrypted message transfer. If no `ReplyPort` argument is given a per-thread global reply port is used.

2.7.2 Usage Problems and Hints

In [25] Draves identifies problems that became apparent during the long-term usage of MIG. He names the confusion of users concerning the difference between interfaces and implementation.

Another problem identified by Draves is the generation of open-code stubs rather than using closed-code interpretive stubs and generating descriptors for these stubs. Open-coded stubs are not desired when distributing proprietary services, because the communication protocol can be analyzed easily. But, generating open-code stubs cannot be avoided if the generated code should be integrated into the calling client such that the compiler can optimize its usage.

Additionally, Draves also states that the assignment of message identifiers is a problem. There exist different approaches: from generating string-based descriptions, as is done with CORBA, to using integer identifiers assigned by the user in a user-defined include file, as is done with Flick. Optimizing CORBA implementations, such as TAO,

one should use hashing functions to have constant lookup times to match string message identifiers to dispatch functions.

Draves also gives some hints to the users of MIG. Besides others he recommends to avoid asynchronous messages because the complicated protocol to be used in user-space. He also does not recommend the usage of `inout` parameters, that is, parameters that carry data both ways. This is problematic if optimizations should be applied for one direction. DICE tries to solve this by providing hints to the compiler which can be specific for one direction of communication. Draves also suggests declaring used types in separate files so they can be used by different interfaces without the need to import other interfaces. This feature is even more rigorously supported by DICE: it allows including header files which contain type declarations in the target language. Another suggestion is the usage of a MIG extension to differentiate the generated function at client and server side.

An implementation problem of Mach mentioned in [26] is reverse port lookup: A server has to find the reply port for a client request. Using a *send-once-right* for the reply messages allows supplying the server with a reply port easily. Also, the send-once-right avoids multiple replies if the send right is propagated along a line of receivers and only one of them is allowed to reply.

2.7.3 Optimizations

Barrerra III described in [57] two cases for which IPC should be optimized: small messages (about 128 bytes) and very large data. The reason for only these two cases is that applications tend to cache data. Either large chunks of data are exchanged or small (synchronization) messages. For small messages latency is important, which includes the costs to set up the message and decode the message. If the communication layer, i.e., IPC, is fast, encoding and decoding the message is significant. For large messages the throughput is important. Most of the time of large data transfers is spent copying data. Message transfer can be optimized by borrowing techniques from the V System [18], Sprite [82], Amoeba [92], and Firefly [86]. These techniques include context switch avoidance and minimizing copy operations.

In Mach, copy operations are avoided by using shared buffers. Some problems have been identified which are outlined shortly: A single shared buffer for all tasks offers no protection from interfering tasks. Thus a separate shared buffer is required for each pair of tasks, which limits the shared buffer space. To avoid copies from a shared buffer into the device memory, the device memory should be shared with the client. But, the driver is no longer protected from malicious tasks, especially if message headers should be created inside the shared memory area. This also implies a semantic problem: after an application signals “message ready, send”, it may still manipulate the data inside the message. Also, the application will have to copy the data for the message from somewhere in its address space into the shared buffer. Thus, the work of copying is only moved from the device driver to the application.

Druschel and Peterson suggest in [28] a mechanisms for shared memory communication – *fbufs*. The shared memory region has a write-once semantic. That is, after a

buffer has been filled no further modifications are allowed. Only the creator of a buffer has write access to that buffer. All subsequent users have read-only rights to it. If a component wants to modify or add content, a new buffer has to be allocated. To enforce security the notion of *volatile buffers* is introduced. Access to a buffer is revoked after it has been used. This requires pagetable and TLB modifications and is therefore expensive on x86 architectures.

To optimize the performance of communication stubs generated by an IDL compiler Ford, Hibler, and Lepreau describe the usage of interface annotation in [38]. Interface annotations are known from DCE IDL, but the paper investigates the specific performance gain for an adapted version of MIG. The techniques are for instance used for:

- parameter overloading to specialize the description of a parameter,
- user-provided marshalling,
- memory allocation strategies,
- copy semantics or "used by server" (without copy), and
- varying trust parameters (no trust; confidentiality but not integrity; full trust).

Ford et al. also state that the interface description's main purpose is to specify the *network contract* between the client and the server, i.e., what operations can be invoked at a service and which information has to be passed for the invocation. However, in most RPC systems the interface description also contains the *programmer's contract*, that is, how parameters are passed to the stub and who allocates memory for the parameters.

The Mach Interface Generator was one of the first stub code generators used in a microkernel-based system. Many of the mentioned papers describe methods to improve the performance of the generated stubs or the memory consumption. I will discuss several ideas in later chapters and extend them where viable.

2.8 Flick - Flexible Interface Compiler Kit

The group of Jay Lepreau at the University of Utah started research of microkernels by using Mach but later wrote their own microkernel Fluke [40]. Fluke is an interrupt-driven kernel with an atomic API. It can be configured to be process-based. The main difference between these two approaches as described in [39] is that interrupt-driven kernels have *one* kernel stack and all system calls are atomic and can be restarted or rolled back. Process-based kernels usually have one kernel stack for *each* user process. The performance of interrupt-driven kernels is worse on x86 architectures than process-based kernels, because the x86 architecture is designed for the process driven design.

The Utah group also developed an IDL compiler for their microkernel called Flick [29]. Flick has a modular, three layered design: a front-end that parses the IDL file and generates an *Abstract Object Interface* (AOI) representation; the presentation generator that translates the AOI into a *target language presentation* (PRES); and the back-end,

which writes target code from the PRES with *on-the-wire* specific elements. Through the usage of intermediate layers different optimization approaches are possible.

> Flick implements optimizations that are driven by its task domain and delegates general-purpose code optimization to the target language compiler (such as: register allocation, constant folding, and strength reduction).

Such optimizations include for instance that at the server side data is referenced in the message buffer. Also, the message buffer used to receive data on the server side can be reused to hold the data for the reply message. Eide et al. state that the annotation of interfaces with "hints" can lead to performance gains, but the reliance on hints moves the burden of optimization from the compiler to the programmer, and has the additional effect of making the interface description language non-portable or useful only within restricted domains.

Eide et al. identify the problem that distributed applications often require different communication patterns than local applications of dedicated, synchronous services. To tackle this problem they propose the usage of *decomposed stubs* [30], which consist of

1. pickling stubs, which marshal messages (request and reply),
2. unpickling stubs, which unmarshal messages,
3. send stubs, which transmit marshalled messages,
4. server work functions, which handle received requests,
5. client work functions, which handle received replies,
6. continuation stubs, which can postpone message processing, and
7. dispatch function to dispatch messages.

There also exist separate pickling stubs for operation request, operation replies, and exceptional replies.

Send stubs can be used to asynchronously send messages, that is, to not wait for a reply. However, applications (or wrapper libs) have to be written to account for asynchronous communication. Send stubs take two additional arguments: the target of the message and the communication environment. At the receiver's side the client is identified by the reply port. Also, the message has to contain an invocation ID to match replies with requests. These invocation IDs have to be provided by the application.

The *client* and *server work functions* are used to receive pickled requests or replies and do either unpickle the message and call the server function, forward the message, or postpone message handling. Postponed messages can be re-injected into the message queue using *continuation stubs*.

Eide et al. name some application fields for decomposed stubs:

> Pickling and unpickling stubs provide applications with greater control over the handling of messages and enable certain application-specific optimizations that are not possible within a traditional RPC model. For example, a message can be pickled once and sent to multiple targets; also, common replies can be pre-marshalled and cached, thus reducing response times: An especially useful optimization for Khazana[1] is that a message can be redirected to another node without the overhead of decoding and re-encoding the message (assuming that the message data is not needed in order to make the forwarding decision).

In the context of a multi-tier architecture of servers, that is, servers using other servers to provide their service, as described in Section 3.2, the usage of decomposed stubs seems appropriate. The servers, we wrote for DROPS, with such a multi-tier architecture, all had to manipulate the data they received before propagating the request to the next server. Typical parameters to be manipulated are session identifier and offsets into buffers. Because this requires manipulation, i.e., unpickling of the message, a direct forwarding cannot be applied.

Another scenario mentioned is the preparation of a default answer, which can, for example, be sent by the server in case of errors. However, the decision if a default answer can be sent is made in the server implementation functions. By then, a request has been dispatched and unmarshalled. To speed up this case, the decision when to send a default message has to be made sooner, for instance, directly after receiving a request. This requires a precise description of a faulty message and the layout of the reply message. Such a description will complicate the interface definition and is contrary to the concept of decomposed stubs.

Also, decomposed stubs allow receiving a message and, without unmarshalling it, propagating it to the next component in the processing chain. Similar functionality is provided by DICE: marshalling and unmarshalling functions are already generated as well as generic message receive functions. Send functions that take a pre-marshalled message buffer are provided in a generic library.

Another optimization technique implemented in Flick is the usage of *chunks*: For every group of similar parameters an extra chunk in the message buffer is used. A chunk allows accessing the elements of the message buffer using fixed offsets to the start of the chunk.

Eide defines in [31] three requirements on software development tools for distributed applications, esp. IDL compilers.

1. minimize human effort required to design, implement, and maintain the (distributed) application
2. result in efficient and fast application code
3. support the application's overall design and programming model

[1] a "global memory service" application based on the Fluke microkernel

Gefflaut et al. described in [44] their application of Flick to the construction of an L4-based multi-server operating system. Their conclusion was: Flick generated stubs are too slow. This is mostly due to the fact that the generated code does not marshal parameters directly into registers, which would require platform specific optimizations.

As a summary the authors write:

> However, it is clear that an improved code generation facility has to be developed that generates near-optimal code.

2.9 Concert

This section introduces an approach to interface specification by annotating the primary programming language. The argumentation for the Concert language in [9] is: most programs or libraries already contain some sort of interface specification, which are specified with *interface definition sub-languages* (IDS).

The IDS for the C language is, for instance, the collection of type and function declarations in a header file. To extract a unambiguous message format from a C function declaration it has to be annotated. These annotations are equal to IDL attributes and denote the direction of the parameter transfer, maximum length of arrays, and similar information.

In [8] Auerbach and his colleagues name some advantages of the Concert approach: Concert has a one-translator compilation compared to two-translator compilation of IDL-based systems; also, less annotations are necessary, because obvious annotations are default; and, the developer does not have to learn a new language, but can write the IDS in the language he or she is used to.

The usage of IDS was not applicable to our system. The programming language used the most in our group was C. C allows too many ambiguities. A pointer to an integer, for instance, can describe a variable sized array of integers or simply the memory location of an integer. An annotation of these widely used types was necessary. We also used a C compiler that could not easily be modified. Instead a second, pre-compiler, would have to be used, which violates one of the arguments of the Concert authors.

2.10 Spec#

Another approach to enrich the target language with additional information is the Spec# programming system, which is built on top of Singularity [56, 34]. In [10] Barnett et al. describe the usage of annotations in the programming language to define contracts as interfaces between components. The main intention of Spec# is to verify the correctness of a program by using the annotation to build run-time checks into the program or, if possible, perform compile-time checks. The authors state that “to best influence the process by which a software engineer works, one can aim to enhance the engineer’s primary thinking and working tool: the programming language.” Spec# aims are uniting informal documentation and interface description in the programming language.

Communication code is automatically generated from the description in Spec# by the Spec# compiler. In [11] security issues with communication are discussed. More security implies a more complicated security model and more copy overhead (integrity and partner verification) during communication. In Singularity [56, 34] data is exchanged over channels with exactly 2 endpoints where one endpoint can at any point in time be owned by only one thread. Memory buffers can be transferred by pointers, with the semantics that the ownership of the memory is transferred. Channel communication is governed by statically verified channel contracts, which describe the messages, message parameters, and valid message interaction sequences. To establish communication channels, channel endpoints can be parameters of a message. Also, sending and receiving requires no memory allocation and sends are non-blocking and non-failing. By verifying that a sender does not access memory which it no longer owns, the Spec# compiler can improve security.

The Spec# approach is also not viable, for reasons mentioned before: The main development language in our group is C, which is due to the heavy reuse of open-source code from the Linux kernel. Using a new, secure language would have required to write many drivers from scratch.

2.11 Barrelfish

Baumann et al. make a strong case in [13] that future operating systems will be structured as distributed systems. They argue that future hardware will contain many CPUs of different types accessing different memory. Because of the heterogeneous nature of this hardware, simply SMP operating systems won't suffice. Instead many operating system kernels run of the different CPUs. To access the services provided on other CPUs, the kernels send messages as only means of inter-kernel communication. Barrelfish uses a derivate of URPC as communication mechanism [12]. URPC [15] uses shared memory and user-level thread management to prevent costly invocations of the kernel. I will show that shared memory communication is only one special form of communication. For rights delegation and memory management the involvement of the kernel is inevitable.

2.12 CORBA

In [46] Gokhale and Schmidt analyze the single parts of generated and hand-written communication stubs to identify bottlenecks. For client side stubs they state: "The analysis of the performance of the Common Object Request Broker Architecture (CORBA) versions suggests that presentation layer conversions and data copying are the primary areas that must be optimized to achieve higher throughput." Presentation layer conversion can be ignored in IDL compilers for microkernel-based system, because communication on the local node does not alter the presentation of the data.

At the servers side the CORBA stubs performed multiple dispatching steps: find the object and the to-be-called method in the object. The analyzed CORBA implementa-

tions used strings to specify object type and method. Thus expensive string comparison is made while dispatching. As an alternative a numeric dispatching is suggested. This behavior is implemented in Flick, `sunrpc`, and others. Also, CORBA specifies a variety of services that should be used and implemented by an IDL compiler and the generated code, such as naming service or object creation service.

The TAO real-time object request broker and its IDL compiler provide some optimizations to speed up the communication and dispatching. One optimization is using a hash function to provide a constant-time lookup of the object and the respective method. Another one is the support for *static invocation interface* [47] as opposed to the CORBA default of *dynamic invocation interface*. Clients use the static invocation interface, i.e., static stubs, if they know at compile time which object and which method to call. Dynamic invocation interfaces, on the other hand, are used when the caller has no compile-time knowledge of the methods it invokes.

To increase flexibility of applications using CORBA, the TAO IDL compiler also supports Asynchronous Message Invocation (AMI) as described in [7]. The AMI callback model is defined in the CORBA Messaging Specification [49]. It specifies an alternative implementation for client stubs in which a client may send a request to a server and asynchronously handle the reply. In this scenario, the server's implementation is left unmodified. There are two ways to handle the asynchronous reply: polling and callbacks. By polling the client regularly checks for the reply. With callbacks, the client has to provide an ORB which handles the reply message from the server and dispatches the message to the appropriate callback function. To identify replies from different servers Asynchronous Completion Tokens [84] are used. That is, the request to the server has an additional `inout` parameter, the ACT.

2.13 Peregrine

In [59] the Peregrine high-performance RPC system is introduced, a system for distributed and local communication. The paper describes several performance optimizations integrated in the Peregrine RPC system.

- **copy avoidance**: this is achieved by mapping the message into the kernel and either using scatter-gather DMA to transfer the data directly via the network card for remote communication or copy it into the receiver's message buffer for local communication. Mappings on the described target platform (Sun Microcomputers) are cheap.

- **avoid data conversion**: when client and server set up the communication session the server transmits its data encoding format. The client library can then decide whether data conversion is necessary or not. This is an enhancement compared to `sunrpc` where data in a message is always converted to or from the external data representation.

- **"pre-compiled" headers**: on session setup the message headers for Ethernet and IP protocol stack are configured and reused for subsequent calls.
- **reuse of client call stack in server**: The message buffer is built around the parameters placed by the compiler on the client's stack when invoking the client stub. At the server the message buffer is mapped into the server's address space at a memory page boundary and the server thread is started with this memory page as stack. This way, the server function can reuse the client's call stack.

Some of the optimizations mentioned cannot be transferred onto our most common target architecture, the x86 platform, because mappings are expensive. On x86, the remapping of a memory page includes flushing the translation look-aside buffer (TLB).

2.14 IDL Compilers for L4 μ-Kernels

There exist several IDL compilers for L4-based systems, which I will shortly introduce in this section.

One of the first IDL compilers for L4-based systems was a modified version of Flick. As already described in Section 2.8, Gefflaut et al. describe the impact of the generated stubs on the overall performance of the Sawmill system. As a result of that work I developed an IDL compiler for the L4 implementation Fiasco. Parallel to my work Andreas Haeberlen started to work on IDL^4 in Karlsruhe for their L4 implementations Hazelnut and Pistachio. After the group at University of New South Wales started to use the L4 kernel from Karlsruhe they also wrote their own IDL compiler, called Magpie.

2.14.1 IDL^4

In [51] Haeberlen writes "Because this work is targeted at multi-server operating systems (and not distributed systems), we assume that interoperability is not required, i.e., that a client and server run on the same machine, use the same kernel, and that both stubs are compiled with the same IDL compiler. For the same reason, we assume that no messages are damaged or lost by the underlying transport mechanism (in this case, RPC). Finally we restrict ourselves to static invocation, i.e., all interfaces must be known at compile time. This permits us to use a fixed message layout and eliminates the need for tagging elements."

These assumptions allowed Haeberlen to integrate some extensive optimizations into IDL^4, such as direct stack transfer [52]. The sender's stack is transferred as message buffer to the receiver, which then uses the stack to invoke the server implementation. Because client and server use the same target language compiler the same stack layout can be assumed. Here the stack is copied, as opposed to Peregrine, where it is mapped.

2.14.2 Magpie

Magpie [37] is a template-based IDL compiler. It uses templates written in the target language and replaces placeholders with values from the IDL. This approach allows to

easily change the target code. But it also removes some degree of freedom from the user. The generated code always has to conform to the template. Optimizations, such as using inline assembly code, require an additional template. For multiple features, multiple templates have to exist or should be combinable. This makes the template-based IDL compiler complex.

Magpie achieves some of these features by writing the templates in an interpretive programming language. Parts of the template can be replaced with code snippets and the template itself is executed. This allows to recursively apply transformations. As mentioned in [95], changes to the IDL syntax require changes of the templates *and* the IDL compiler.

2.15 DCE++

In [80] Mock extends the DCE communication model, which is a strict client-server model, to a coherent object model. In a client-server model, the server manages multiple objects via one interface. The addressable unit in DCE is the server. Thus different clients using different objects all refer to the same unique entity. Mock proposes the extension of the C-based DCE concepts to object models, where the addressable unit is an object.

Also, methods to transparently migrate objects in a distributed environment are discussed. The explicit notion of a node is introduced. The caller will still be unaware of the locality of the called object. But objects may request other objects to migrate so communication will be more efficient.

Even though the addressable unit is no longer a server, but an object, the mechanisms to communicate with the objects stay the same: remote procedure calls.

2.16 Heidi

Interface Definition Languages, especially CORBA, have a strict specification of its language mappings, that is, how the IDL is translated into the target language. If interfaces should be integrated into existing applications, the defined language mapping is often not compatible with existing code. Therefore, it should be possible to customize the language mapping.

In [95] Welling and Ott propose such a customizable IDL by specifying an own type mapping. They proposed the usage of a template-based IDL compiler for their system Heidi. The compiler will include a mapping of IDL types to target language types. Whenever a target type is inserted into the target code, the respective mapping is selected. This approach allows adapting to new language mappings easily.

An example mentioned in the paper is the naming of classes for client and server stubs. The server's side classes often contain suffixes to differentiate them from the client classes and because there exist more than one server class for an interface. In the example one of the class types can strip its suffix by modifying the mapping used for this particular class.

2.17 Ada

The Ada programming language [91] has some basic integration of communication principles. It contains the explicit notion of tasks and consequently has means to express communication between tasks. Messages are exchanged explicitly by invoking procedures in other tasks. The communication partner can wait for message invocations using the `select` statement. With this statement it can wait for one or more procedure invocations or for timeouts. One specialty of Ada is the combination of the execution of a message with preconditions. E.g., a `select` statement can wait for the arrival of a number of messages. Only if all messages have arrived, the procedure is executed.

In Ada messages are specified in line with the rest of the code. The compiler transparently generates the communication code. This is similar to the IDS of Concert I discussed earlier.

2.18 Summary

In this chapter I described different communication mechanisms, such as MPI and RPC. I introduced different projects and their struggle with the same problems in the communication stubs: performance and memory consumption. Several approaches were discussed, such as static versus interpretive stubs or copy avoidance. I also presented different approaches to specifying the contracts for the communication between client and server.

The main focus of this Thesis being the communication in microkernel-based systems, I focused on the discussion of stub code generators for microkernel-based systems (MIG, Flick) and especially L4-based systems (IDL^4, and Magpie).

Chapter 3

Forms of Communication

Liedtke motivated in [53, 67], that microkernels should be small and fast to allow reasonable performance for applications running on top of them. One major implication of his work is that the interface provided by a microkernel contains only mechanisms but implements no policy. Every service that could be implemented outside of the kernel should be. As a consequence of this minimalism, mechanisms with similar functionality have been combined to reuse code and keep the size of the kernel small. These mechanisms include communication, receiving interrupts, and memory control. User applications running on L4 use the communication mechanism to implement synchronization or notifications. In addition to that, shared memory is used to reduce copy operations of large data.

In this chapter I discuss all these different forms of communication starting with simple message transfer, asynchronous communication, up to shared memory communication. I will then demonstrate how each can be supported by an IDL compiler.

3.1 Message Passing

Previous work on communication in distributed computing environments was mostly focused on providing transparent remote procedure invocations. That is, the invocation of a procedure on a remote node appears as if it is made locally. The procedure invocation is synchronous, i.e., the invoker waits until it receives a reply from the remote procedure. Research on remote procedure calls (RPC) focused on error detection in message delivery and transparent data conversion between representations in the sending and receiving node.

To standardize the communication in distributed networks two main efforts were initiated. The Distributed Computing Environment [50] was developed by the Open Group (formerly Open Software Foundation) and the OMG developed the Common Object Request Broker Architecture (CORBA) Standard. Both allowed invoking remote procedure calls at service providers, which could run at any arbitrary location, including the same node as the caller.

In addition to RPCs, the concept of *one-way* messages exists. *One-way* message

means no return parameters. Yet, the delivery of the message has to be acknowledged in the communication layer.

As mentioned above, simple messages can be means to synchronize multi-threaded operations or to signal events. To support these two uses of IPC communication in a microkernel-based system, I applied the concept of *one-way* messages to an IDL compiler.

Remote procedure call (RPC) communication is service centric. A client and a service are not equal communication partners. To receive messages in a RPC scenario, an application has to implement a service. I discussed this restriction in [4] and proposed the usage of servers not only as message recipients but also as message sources. I introduced the notion of an `out` attribute which is the counterpart of the one-way message. The Microsoft IDL (MIDL) [74] provides the notion of the `callback` attribute that can be associated with functions. In contrast to the `out` attribute, it does define a static function at the client that can be called from the server with an RPC whereas the `out` attribute specifies simple messages that can be emitted by the server. Notifications to be sent from a server to a client are implemented by letting clients wait for a message from the server. If multiple clients wait for the same message, the server iterates over the waiting clients and sends a reply.

The usage of the `out` attribute in our group showed, the concept is not intuitive to developers. A server implementing an interface is usually the *sink* of messages not the *source*.

Let us look at an example from the Comquad project [19]. We implemented a resource reservation framework where resources are reserved by clients using the *reservation interface* of resource managers. In our design the availability of a resource may change. For instance, an application with high priority requests resources, which have to be revoked from a lower prioritized application. We could have used the `out` messages of the resource manager to signal the change. However, we decided to define a notification interface, which each client has to implement. Thus the client becomes a server itself, even though the notification interface only provides one function.

Due to the structure of service providers, each server has to implement the marshalling and unmarshalling code for the messages, a server loop that waits for all possible new messages, and it has to implement a dispatcher that dispatches the messages. For a client with one function in the *notification interface* this is undesirable overhead. More elegant is the specification of an `out` function in the interface of the resource manager. A client then waits explicitly for this message.

Combined with *one-way* messages the `out` function can also be used without server loop and dispatch function. This allows the reduction of RPC to simple message passing primitives.

Simple messages are *not* equivalent to the interface of a service, but basic blocks of construction for an interface. The combination of an `in` and an `out` message form an RPC (reply comes from the destination of the request). The combination of one or more RPCs with a server loop and a dispatch function create a service.

Due to the restrictions of the target language C[1] the receipt of multiple `in` or `out`

[1]The C and C++ programming languages are the main programming languages for operating sys-

messages, requires more consideration. To receive only one message `A` we can provide one function `recv_A`. If we want to receive either a message `A` *or* a message `B` then we need a function `recv_A_or_B`. Because we also want to receive only `A` or only `B` at other places, the functions to generate sum up to three. For an additional third message `C` an IDL compiler has to generate six receive functions.

To minimize the number of generated functions, the DICE IDL compiler generates one function to receive a specific message (e.g., `recv_A`), one function to receive any message (`recv_any`) and unmarshalling functions for the specific messages (e.g., `unmarshal_A`). This way the number of generated functions is reduced from $n!$ to $2n + 1$. Also, the functions can be used to receive any message and decide what to do: discard the message, send error reply, or unmarshal and process the message.

With a different motivation–asynchronous communication–, the operating systems group around Eric Eide and Jay Lepreau designed *decomposed stubs* [30] that can be used to do similar message transfers. Decomposed stubs are described in Section 2.8.

If only one message can be received at any given time, the operation identifier is redundant. To disable its usage the `noopcode` attribute can be assigned to an operation. Also, the `noexception` attribute can be specified to avoid the transmission of system exceptions from the server to the client. Exceptions and opcode occupy a word in the message buffer which can be used for payload.

In summary, one-way messages can easily be integrated into an IDL compiler using attributes. They allow the exchange of signals or to synchronize components. I described the complexity of distinguishing multiple messages in a programming language, such as C. I further introduced solutions to identify the correct message at a receiver and how to minimize a message in case a distinction of messages at the receiver is not required.

3.2 Asynchronous Servers

In microkernel-based operating systems different services reside in different protection domains (address spaces). A higher level service, such as a file system, may require functionality of a lower level service, such as a disk driver. The latency of such higher level services depends on the latency of the lower level services. In a strictly synchronous setup, the higher level service will not be able to process new request during the time it idly waits for the lower level services.

To illustrate the problem, let me highlight the L4 version 2 and version 4 APIs. The target of an IPC is identified by a global thread ID. It is not possible to let multiple threads wait for requests to the same service. Instead a service thread has to receive the requests and propagate them to the worker threads. The service thread then waits for the next request and delegates it appropriately.

A special problem of the L4 version 2 API is that the invoker of an IPC call expects the reply from the same communication partner. If a service thread forwards a message to a worker thread, this worker thread cannot reply to the client. The service thread has

tems. Therefore, these two programming languages should mainly be supported by an IDL compiler for microkernel-based operating systems.

to collect the replies for the original requests and send them back to the clients. It has to provide entry points for these reply notification from the worker threads. Because these functions can be derived from the original interface operation, the DICE IDL compiler generates these functions automatically if the `allow_reply_only` attribute is set for a function.

This mechanism can be used to solve the above mentioned problem of a higher level server depending on a lower level service. If a request is forwarded to a lower level service, the state of the request can be stored. The processing function calls the lower level service asynchronously and then returns to the server loop where it can wait for (and process) new requests. When the lower level service replies, the original request is continued and a reply is sent to the client. I call these services *asynchronous servers*.

Another use of this feature is the possibility to indicate which function in a multi-threaded server should run in its own thread. Whenever such a function is called, the generated code starts a new thread and executes the requested function. The thread will signal the service thread its completion, which then terminates the thread and sends the reply to the client. This approach has some downside: firstly, the generated code only starts and stops threads. It does not reuse existing threads. Secondly, the generated code relies on a thread library. The dependence on external, i.e., not generated, code should be avoided for a code generator for microkernel-based systems.

To loosen the dependence on external functionality I use a proxy library with DICE. As an example: Instead of using the C library function `malloc` the generated code uses the function `_dice_malloc`. This function is implemented in the DICE library and defined as a weak symbol. This allows users to provide their own implementation of `_dice_malloc` and link this implementation to the generated code.

With the mentioned proposal of creating threads for single functions of a service, the thread library has to support starting functions with different numbers and types of arguments. As a compromise to automated starting and stopping of functions I defined the function signature of a service dispatch function to match with the start function required by our thread library. A developer can define an interface for one or more functions that should be combined in a worker thread. In one of the original component functions a new thread may be started with the new dispatch function as start function. This gives the developer the freedom to decide when to start a new worker thread. It also allows intelligent thread management.

Asynchronous servers solve some of the restriction of the synchronous IPC mechanisms provided by the L4 microkernel. The functionality of this feature has been balanced to give the developer the freedom to decide when to use it and the impact on performance of the generated stubs. Support for the generation of notification messages is achieved with the `allow_reply_only` attribute, as well as worker threads, which can be created using a nested interface and its dispatch function.

3.3 Shared Memory

When data is copied between protection domains, it is copied from the source location into a temporary storage in the kernel memory region and then to the destination location. For large amounts of data, these copy operations make up most of the communication time. To make the communication faster, copy operations should be avoided. The L4 microkernel eliminates one copy operation by making the source location visible in the destination's protection domain (inside the kernel memory area). Thus the message can be copied from the source location directly to the target location. However, the cost of the temporary mapping contributes to the overall communication cost. Also, the source of the communication copies the parameters of a message into a message buffer and the destination copies the parameters out of the message buffer.

A mechanism to avoid these copy operations is the shared memory communication. Instead of copying data into the message buffer, then copying the message buffer from sender to receiver, and finally copying data out of the message buffer again, the message buffer can be in a memory region shared by sender and receiver. Access to this shared message buffer has to be synchronized to avoid corruption of data.

A naive support of shared memory communication in an IDL compiler could result in establishing and revoking a shared message buffer for each call from a client to the service. Establishment and revocation of shared memory are costly operations and should rarely be used.

A more feasible approach is to establish a permanent shared memory region between a client and a server. Thus the shared memory region becomes part of a session between a client and a server. To further reduce the number of copy operations, the transferred data should be generated and consumed directly in the shared memory region. This approach requires a different application designs than with common RPC communication.

If the usage of shared memory communication cannot be hidden from the user of a generated communication stub, the support for shared memory communication in an IDL compiler is questionable. Also, using the session-based approach, the server has to establish a session with every new client and manage the associated shared memory region. I instead suggest using RPC to establish a shared memory region and use simple messages to synchronize the access to this region.

To back my conclusions, I will discuss a possible implementation of support for shared memory communication in an IDL compiler. I will enumerate all the consequences for the following example.

In the interface definition language a parameter that should be exchanged via shared memory receives the `shared` attribute. The attribute can only be associated with pointers. Assuming the shared memory region is already established, the generated stub does not contain any copy operations for this parameter, but instead checks if the pointer is inside the specified memory region. It then computes an offset to the start of the memory region and transfers this offset to the server. The server adds the offset to its own start address of the shared memory region and uses the resulting pointer as the parameter.

For instance:

```
interface socket {
  void write ([in, shared] unsigned char * packet);
};
```

The interface `socket` consists of one function that sends data referred to by the parameter `packet` to the server via shared memory.

The IDL compiler generates the following functions for this interface at the client side:

- `socket_shared_init` – creates the shared memory region
- `socket_shared_alloc` – allocate memory for parameters in the shared memory region.
- `socket_write_call` – the original communication function for the `write` operation from the interface
- `socket_shared_free` – free the memory occupied by parameters in the shared memory region.
- `socket_shared_close` – revokes the shared memory region

At the server side, the following functions are generated in addition to the functions already generated for conventional communication:

- `socket_shared_free` – free memory allocated for the parameters in the shared memory region.
- `socket_shared_alloc` – allocate memory in the shared memory region for parameters transferred back to the client.

Functions to establish the shared memory region are not generated for the server. Instead the code is directly generated into the existing server loop.

To allow the use of different shared memory regions for different parameters the `shared` attribute may contain an identifier: `shared(<id>)`. This identifier is used to suffix the special functions for the shared memory region: `socket_shared_<id>_init` etc.

Existing programming models already have data in private memory and have hand it off to the communication stubs. If a client wants to transfer a parameter in shared memory, it has to allocate the argument in the shared memory region *before* filling it with data. The programming model has to change to support shared memory communication. An example for this programming model is the use of `sk_bufs` in the network stack of the Linux kernel.

Most of these functions, especially the establishment of shared memory region and the allocation and deallocation of data in that region, are the same for different interfaces.

They differ only in their name and the size of the shared memory region they establish. Thus, these functions can also be provided by a library.

Taken all these caveats into account, the integration of shared memory communication in this form into an IDL compiler is not justified. Instead a library can be used for managing memory in a shared memory region. Establishment of a shared memory region can be done using communication stubs generated from an IDL. Notification messages can also be represented in an IDL. This follows the minimalistic L4 principals: Only mechanisms to establish a shared memory region are provided; The policy how to manage that region is left to upper layers.

3.4 Streaming

Another form of communication is *stream* communication. Here, no concurrent modification of the same data is performed, but rather data produced by one application is placed into a buffer and consumed by the other application. The data is exchanged in *packets* of the same size. The buffer may contain one or more of these packets.

Usually, the buffer also contains some administrative data indicating the next free packet for the producer and the next packet to be consumed. It also contains the two counting semaphores that are used to keep track of the number of free and occupied packets. Previous work by Löser and Reuther [71, 72] provides an in depth discussion of an implementation of a streaming interface on top of the L4 microkernel.

When supporting stream communication with an IDL compiler, the basic concepts are the same as for shared memory communication. The parameter to be transmitted has to be attributed. The packet to be transmitted has to be allocated in the shared memory area and the invocation of the generated client stub will commit the packet to be consumed. The server side generated code gets the available packets from the shared memory region and after they have been processed, frees them.

Similar to shared memory communication, it is not transparent for the user to use stream-based communication. A server and its clients have to be specifically designed for this sort of communication. Therefore, it seems unfeasible to integrate the support of stream-based communication into an IDL compiler.

3.5 Security

Bishop describes in [17] the three main aspects of computer security as *confidentiality*, *integrity*, and *availability*. *Integrity* refers to the trustworthiness of data or resources, and it is usually phrased in terms of preventing improper or unauthorized change. *Confidentiality* is the concealment of information or resources. *Availability* refers to the ability to use the desired information or resources.

In the context of communication on microkernels, we can assume that some aspects of security are always fulfilled. The microkernel API ensures that sender and receiver are authentic. That is, sender and receiver cannot fake their IDs. Also, changing the

transmitted data is not possible. With respects to confidentiality I will explain possibilities for access control to services. I will also discuss the concealment of services as a whole. I will however not discuss how to conceal the resource usage of a server or other covert channels. With respect to availability I shortly discuss *denial of service attacks* and *robustness* but I will not discuss aspects of reliability.

3.5.1 Access Control

The L4 microkernel API versions 2 through X.2 specified that threads are addressed using global thread IDs. Even though one can hide mapping of a service to a thread ID by restricting name services, malicious application may guess thread IDs. A possible solution is to build systems with process local communication endpoints and thus denying malicious applications the ability to guess thread IDs [61, 42, 22].

Another implementation of access control is to intercept messages to an application using microkernel mechanisms. The messages are redirected to a communication monitor [58, 32]. This monitor can then either propagate the message to the real service or deny the communication. In the context of service level access control this method is very coarse grained. Services often want to control access based on a {subject, object, operation} matrix. The matrix specifies which operations are permitted for which subject on which object. Access control based on communication control can only allow or disallow all operations on all objects by a specific subject. The described methods are useful to hide services from clients, but not to do fine-grained access control.

An example of one thread providing multiple, distinct services is the implementation of a dynamic application loader for the Dresden Real-Time Operating System. The loader provides an interface to start tasks. Access to that interface is restricted to applications with the right to start new tasks. When the task starts, the loader will also be the memory manager (i.e., pager) for the newly started task. This implies that the newly started task requires access to the functionality to map pages. Because the services to start an application and to provide memory for a newly started application are implemented in the same thread, the newly started application implicitly gains access to the task management interface, even though it may not be eligible.

Access control to objects has to be performed at a higher level than the microkernel. One possible spot to control access to an object is the generated server loop of the service. The server loop can check incoming messages for a valid sender address and drop, reject, or allow the request. Using the described access control matrix (allowed operations on object by subject as described in [48]) a callback routine is called from the server loop. This callback routine receives the sender's ID, the message buffer, and the desired operation as parameters. Missing from the parameter list of the access check function is the object identifier. Because this could be anything from an integer value as the first or last parameter, over a string identifier, up to nothing for generic functions such as `open`, this information cannot be retrieved in a generic server loop. To allow the callback routine to obtain object information the message buffer has to be passed as parameter. However, the internal format of the message buffer is opaque. Therefore, this approach is impractical.

Access control is bound too much to the specific implementation of a service: every application might call, for instance, operation f on any object. Operation g may only get called by a subset of tasks, e.g. all tasks with a task ID greater than $0xa$ and less than $0x2f$. Operation h might even be restricted to certain thread IDs. For applications calling operation i thread or task IDs are irrelevant for access control, but only a subset of parameter values is allowed. And a combination of task ID and parameter values may be required for operation j.

Therefore, the most practical solution is the implementation of the access control in the server's component functions. All parameters are available including the object identifier (whichever parameter that currently is) and the sender ID. Because access control at this level is implemented by the component programmer, it is not generated by the DICE IDL compiler.

To support some access control on communication level even on L4 APIs with global thread IDs, I introduced an attribute called `dedicated_partner`. This attribute can be associated with an interface. The server loop will then wait for IPC only from one specific thread. This allows session-based services to establish a closed connection between a worker thread and a specific client.

3.5.2 Shared Memory Communication

When using shared memory communication, the participating parties have to trust each other regarding the content of the shared memory. Because of the common mistrust regarding private data, it is not desired to retype an application's private memory into shared memory. The granularity of the memory sharing is often larger than the size of the data to be shared and sensitive information may be leaked. Also, copying data from a private memory location into the shared memory location negates the intention of the shared memory communication: copy avoidance. Therefore, data to be shared should always be allocated in the dedicated shared memory region.

When using shared memory communication, we have a chain of applications in protection domains processing the same data. Because the data processing is sequential, only one application at a time should have write access to the data. *Fbufs* have been proposed in [28] as a solution to this problem (also refer to Section 2.7.3). The authors suggest mapping the shared memory region writable to only one application. When the next application should process the data, the right to write the shared memory region has to be transferred. On the x86 architecture, this requires manipulation of at least two address spaces and TLB flushing, which is a costly operation.

Drushel and Peterson's assumption, that the same data is processed by more than two applications in a chain, is true for network requests, where the data to be transmitted is propagated between the components of the network stack. Only additional information is added. Other application scenarios of shared memory communication do manipulate data in every step. Video processing, for instance, alters the exchanged data when demultiplexing and decoding. For these sorts of applications, simple shared buffers between two partners are sufficient, where one application has write access and the second application has read access.

3.5.3 Robustness

Providing a robust service can, for example, be achieved by ensuring that a service is robust against invalid parameters. This can be enforced by checking valid values or value ranges. To help the programmer with this validation, parameters in an IDL file can be annotated. However, this approach will complicate the description of a constructed type. Also, the validity of one parameter may depend on the value of another parameter. A verification of parameters is simpler if the component function tests the parameters itself. The only test currently implemented in DICE is for NULL pointers and the maximum size of buffers.

A different kind of robustness is achieved by broadcasting a request to multiple implementations of the same service. The received results can then be compared and wrong results can be discarded. Such a wrapper could be generated by an IDL compiler but implies some problems. If one implementation of the service is corrupted and will not send a reply, the client wrapper will wait forever for the last outstanding reply. To be able to live with corrupted services a threshold could be specified, for instance, 70% of the requests have to be answered. To avoid stalling servers when sending replies to clients, servers send replies with send timeout zero. If a server sends a reply with timeout zero, the reply may get lost, because the client wrapper is busy sending requests or receiving the reply of another service. To fulfill the required threshold, servers have to reply with (potentially infinite) timeout. Because the wrapper will return to the caller when it received exactly 70% of the replies, following replies from servers will be blocked unnecessarily. Also, the wrapper might wait forever, because only 69% of the replies arrive, but the rest was lost. Thus, the wrapper has to combine a timeout with the threshold: wait for replies until timeout or 70% of replies arrived. The latter approach will still block servers above the threshold unnecessarily.

All these limitations make it hard to find a generic description and implementation that suits every user. Therefore, it is left to the developer to implement such feature.

3.5.4 Denial of Service

In [70] Liedtke et al. describe different denial of service attacks on microkernel-based systems. They identify IPC as the only mechanism provided by the L4 kernel to be used for Denial of Service attacks. For servers which use *open wait* to wait for incoming requests they identify *pulsar attacks* as problem. The server is attacked by clients that bombard it with junk requests. As countermeasure dedicated threads are recommended, that is, server threads, which only service one client (refer to Section 3.5.1).

For servers that have to use *open wait*, it is recommended to limit the servers buffer size to limit the time to identify a junk request and thus to limit the time which the server is not available. DICE generates messages buffer with a size that all parameters just fit in (at the receiver's side with a maximum size of the variable sized parameters). This still may be too large. A countermeasure could be to cascade requests and make the initial request contain the operation identifier only. The decision if a request is valid can be made faster, based on the operation identifier. If a valid operation identifier is

received, the server may then wait for the actual message from the requesting thread. This again limits the size of the receive buffer. However, such servers can be attacked more efficiently: a malicious client sends an operation identifier but not the actual message. The server will block indefinitely.

When using session-based servers, the server may wait only for *open session* requests. Subsequent requests are sent to a dedicated worker thread. A denial of service attack could simply open unused session and thus consume resources. This could be restricted by allowing only a fixed number of concurrent sessions per client. Determining a maximum number of sessions or using a different algorithm to restrict access to a service can be implemented in a component function and does not require support from a stub code generator.

Another attack scenario is the specification of a message buffer in an unmapped memory region of the sender. During IPC (after the rendezvous) the kernel will try to access the message buffer, raise a page fault and the sender should service it. But it is malicious and ignores the map request. Thus, the receiver is blocked forever. Countermeasure is suggested to set a timeout of zero for send pagefaults, which is the default for DICE generated server loops.

One more attack scenario is mentioned in the paper: attack on secondary resources– resources used by a server. But that can only be handled by carefully designing the resource managers, which is out of scope for the work described in my Thesis.

3.6 Summary

In this chapter I introduced different forms of communication, which are used in a microkernel-based operating system. I also showed how these different forms of communication can be supported by a stub code generator for a microkernel. This included the ability to use a generic communication framework, yet still exploit platform specific mechanisms.

Chapter 4

Portability

Using an IDL compiler to generate the communication stubs in a microkernel-based operating system has, besides the three requirements mentioned in Section 2.8, another main motivation: to introduce a level of *abstraction* into the communication, to gain independence from the communication platform. This allows migrating applications easily to different platforms by using a different back-end of the IDL compiler.

The benefit of generality – in this case the ability to deploy to different target platforms – often implies the inability to use specialized features of a single target platform. This is addressed by applying target specific optimizations to the generated code. The IDL compiler can generate highly optimized code for a specific platform. This is especially beneficial for users who are not able to exploit these optimizations otherwise. The unification of platform independence and specific optimizations will be discussed in this chapter.

The Concert programming language (refer to Section 2.9) collocates annotations necessary to generate RPC stubs with the target programming language. The main argument for this approach is that the programmer does not have to learn a new language for the interface description. However, the effort to provide support for another programming language, e.g. C++ additionally to C, is unequally larger for Concert than for a standalone IDL compiler. For Concert a new parser for the target language has to be written, a new set of implicit rules has to be defined, and different annotations have to be specified and supported. Our experience shows that implicit rules, which should simplify annotations, according to Concert, are rather confusing. Different developers tend to assume different implicit rules.

The concept of using interfaces in software design has been widely accepted to provide structured software. Designing an interface allows to clearly define the communication contract for an application, which will implement the interface. Using an interface allows to provide more than one implementation for an interface. Making this step in software development obsolete by collocating annotations for remote procedure calls with the target language negates the effort in structured software design.

4.1 Platform Independence

As we mentioned before, microkernels have to be small to be competitive with other kernel designs. This reflects in the size of the microkernel's API. If a new feature is supposed to be added to this API, the whole API has to be redesigned such, that the overall size stays the same. Otherwise the microkernel would no longer be a *micro*kernel. As a consequence, applications have to be adapted to these changes.

When comparing the API of a microkernel with the API of a monolithic kernel, the functions' signatures of the latter are rather static. When new features are added, these features are appended instead of redesigning the whole API. This is necessary to provide compatibility to existing applications, which depend on the API.

As a result, the API of a microkernel changes fundamentally more often than the API of a monolithic kernel. One example: the first version of the current L4 microkernel was specified in 1996 [66]. Up to now, in over ten years, the specification underwent several changes. Because of the aforementioned reasons, major changes expressed themselves in new versions of the kernel API. These include L4 version 2.2 (also dubbed LN) [68] in 1998, L4 version 2 for MIPS [33] in 1999, L4 version X.0 [69] in 1999, L4 version X.2 [20] with revision 2 dating May 2003 up to revision 6 dating November 2006, L4 N1 [93], L4.Sec [62], and seL4 [21] all in 2005. In the same timespan Linux had (only) 3 major version changes from Linux 2.0 (1998) to Linux 2.6 (2006). The L4 API changed between the different versions dramatically whereas the Linux API was kept relatively stable.

The implementation of the L4 version 2 API here in Dresden–*Fiasco*–has undergone several extensions, including kernel info-page (KIP) system calls (2003), real-time scheduling (2004), absolute timeouts (2004), and IPC monitoring (2006)–to name a few. These extensions have always been a tradeoff for some other feature, for instance, using chief bits in the thread ID for absolute timeouts, or have been carefully interwoven with existing features, such as using the switch-to system call for real-time scheduling.

API stability for applications in microkernel-based operating systems is provided at a higher level–the system services. The portability of applications is achieved by using IDL for communication interfaces. The IDL compiler then "hides" the differences of microkernel APIs. In [45] Murray et al., state that "[DICE] raises the possibility of portable applications running on both Xen/MiniOS and L4/Fiasco." Still, the generated code should exploit available kernel features efficiently.

To achieve the independence of the actual microkernel API but still allowing the programmer to influence the generation of the code such, that it best suits her requirements, we have to identify platform dependent features. If a new platform dependent feature has to be added to the interface definition language, we have to verify, whether this feature could be realized using already existing features. For example, the `ref` attribute, which exists in the DCE IDL, is used to indicate the transmission of an indirect part, which is an L4 specific feature. The IDL compiler can use these hints where appropriate or ignore them. The support of the DCE IDL for platform specific annotations in the form of Application Configuration Files (ACF) was one of the arguments to choose DCE

IDL as the primary interface definition language for DICE.

4.1.1 Platform Dependent Types and Platform Independent Attributes

As an alternative to annotating parameters with platform specific attributes, such as `ref`, platform specific types can be introduced. In the L4 API, a flexpage specifies a memory region using a start address and size. In the following section I will outline the pros and cons of various solutions to support the specification of flexpages in an IDL. Using IDL to express a flexpage the following parameters could be used:

```
[in, size_is(fp_size)] void* fp_start,
[in] unsigned long fp_size
```

The problem with this approach is that different assumptions have to hold for these parameters: the start address has to be page size aligned, and the size has to be a power of 2 of page size. These restrictions have to be enforced in the generated code. Also, the platform specific flexpage type specifies rights for accessing this page, i.e., read-only, read-and-write, or execute rights. It also specifies if ownership of the memory region is transferred to the receiver. To accommodate these additional properties, the specification has to be expanded to:

```
[in, page_size(fp_size), rights(write, own)]
            void* fp_start,
[in] unsigned long fp_size
```

The generated code can then assemble the parameters into the actually transmitted flexpage type. However, this representation includes too much (or not enough) information to be verified, i.e., address alignment, correct size, etc. Instead, an explicit `flexpage` type can be used:

```
[in] flexpage fp
```

Using the platform specific type provides the correct representation for the communication. All the constraints for its members are fulfilled when the surrounding code sets up the argument. On the other hand, using a platform specific type will cause compile errors for target platforms which don't support flexpage types. To avoid these errors for platform dependent types, I chose a compromise between attributing a memory address and a platform specific type:

```
[in, mempage] l4_snd_fpage_t fp
```

Here, a user-defined type, which represents a flexpage on an L4 platform, is used. It is attributed to identify it as a flexpage. On non-L4 platforms the attribute is ignored and the argument is transferred as plain data.

Another alternative is to define the flexpage type solely as a user-defined type with user-provided marshal, unmarshal, allocation, and deallocation routines. This approach makes this type totally opaque to the IDL compiler and deprives it from any possibility to recognize and optimize the usage of a flexpage.

To generalize, platform dependent types are capabilities to kernel-provided objects. A flexpage is the right to access or modify a region in virtual memory. Similarly, a

scheduling context, which can be transferred to a communication partner, is the right to use a slice of computation time. Communication rights are also transferred via communication channels. The code generated by an IDL compiler has to support the transfer of capabilities to kernel-provided objects. The creation and destruction of these objects is not part of the communication.

The generalization of flexpages to capabilities changes the presentation in the IDL annotation to:

```
[in, cap(memory)] l4_snd_fpage_t fp
```

At the server's side a component function is generated, which uses the `l4_fpage_t` type. The compiler has to know that the receiver of a flexpage will only see that type.

Accordingly, the notation for computation time and communication rights can be expressed as:

```
[in, cap(time)] cpu_time_t time
[in, cap(communication)] endpoint_t ep
```

This notation allows to be expanded as needed for new kernel objects.

Currently the Fiasco kernel only allows to specify whether the receiver of an IPC continues to compute on the sender's time slice (lazy scheduling [64, 65]) or uses its own time slices. The former is the default behavior; the latter has to be indicated by setting a bit in the message descriptor. Because every thread can communicate with every other thread, dependencies between threads of different priorities are likely. The communication code makes no effort to avoid possible priority inversion, or other priority related problems.

A first implementation of the computation-time donation used an extra attribute `sched_donate` that can be associated with a function. If present, the generated stubs check a variable in the environment, which is an extra parameter to each generated function. If the variable is set, the mentioned bit in the message descriptor is set. This attribute became obsolete with the generalization of access to kernel objects. For the donation of the current time slice to the receiver of an IPC, aforementioned notation is extended by:

```
[in, cap(time_slice)] boolean donate
```

Discussions with Fiasco kernel developers revealed that an extension of the donation bit is required in the future. This can easily be achieved by adapting the respective code generation and using a type for parameter `donate` that can hold the required values.

4.2 Optimization

One of the main requirements for a tool to be accepted by its users is that it helps them in their existing work flow. Thus, one of the requirements for an IDL compiler for microkernel-based systems is to generate code that is almost as good as hand-written and highly optimized code in relation to size and speed. When comparing the performance of generated communication stubs to the performance of the communication itself, same

sized stubs add more overhead to the fast IPC of L4. Therefore, I had to integrate optimization of the generated code as integral part into the IDL compiler.

Flick, as a different IDL compiler, left optimization to the domain of the target language compiler. This freed Flick from complexity but also led to bad performing communication stubs. In [44] Gefflaut et al. identified the Flick generated code as one major bottleneck in their performance evaluation.

4.2.1 Parameter reordering

To make the generated code faster, the Flick team introduced the concept of *chunks*. A chunk is a group of similar parameters. The beginning of a chunk is stored in a pointer and all members of the chunk are marshalled as fixed offsets from that pointer. This way the target language compiler could optimize access to the message buffer. The generated code marshals the parameters in the order they are specified in the function's parameter list. Thus, if the first parameter is of variable size, e.g., a string, then the string had to be marshalled first and then the start of the next chunk had to be calculated from the length of the string.

DICE generated code reorders parameters. The L4 APIs specify that special message buffer members, such as flexpages or indirect parts, have to be placed at specific locations in the message buffer. For the version 2 API, flexpages have to be at the very beginning and indirect parts at the very end of the message buffer. When packing flexpage parameters for L4 version 2 API, the flexpages have to be terminated by a delimiter–a zero flexpage. The zero flexpage is used by the kernel to stop interpreting word pairs as flexpages.

Inside the normal message part, parameter reordering also allows to sort all fixed sized parameters to the start of the message buffer. Offsets for new chunks only have to be calculated for variable sized members. Also, the reordering packs the variable sized parameters densely.

Let me demonstrate this with an example: A function has as parameters one byte, two short integers (two bytes size), and one word (four bytes size) in that order. When marshalling densely and keeping the order we will see a message buffer as depicted in Figure 4.1[1]. The shown order will have access penalties on x86 platforms, because access is unaligned. For ARM platforms this access is even invalid, that is, unaligned accesses are not allowed.

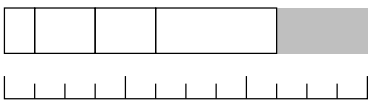

Figure 4.1: Position of parameters. Smallest first, without padding.

[1]The scale indicates the single bytes in the message buffer.

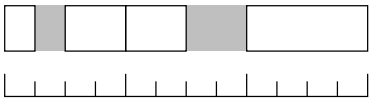

Figure 4.2: Aligned position of parameters. Smallest first, with padding.

To solve the unaligned access issue, the parameters could be padded, so access is always aligned to parameter size. This is shown in Figure 4.2. But this will waste space in the message buffer. By using parameter reordering, we can keep the size minimal and the access aligned, as shown in Figure 4.3. First the word sized parameter is marshalled, then the two short integers at two byte aligned offsets and last the byte.

Figure 4.3: Position of parameters. Biggest first.

Because with parameter reordering parameters can appear in arbitrary order in a message, we need means to specific protocol dependent positions. In the L4 version 2 and version 4 APIs a pagefault message contains the pagefault address in the first word of the message and the address of the instruction that caused the pagefault in the second.

To ensure that certain parameters of a stub will be assigned to the positions in the message buffer defined by the microkernel API, I introduced the `msgpos` attribute. The attribute is assigned to word sized parameters and contains an index into a word array. For the pagefault message of L4 version 2 and version 4 API the IDL looks like this:

```
void pf_handler([in, msgpos(0)] unsigned long addr,
    [in, msgpos(1)] unsigned long eip,
    [out] flexpage page);
```

4.2.2 Copy Avoidance

Another problem of communication stubs is that data is often copied. Usually the generated stubs construct a message on the stack by copying the data from different locations, such as stack, heap, etc., into the message buffer. Then, the message is transferred to the server, which in turn copies the data from the message buffer into local variables. On the return path the same copying happens again. Excessive copying has been identified as a bottleneck before. One approach is to minimize copy operations as much as possible using shared memory communication as described in Section 3.3.

Another approach, which I implemented in DICE, is that large sized parameters, such as strings are referenced at the server side directly in the message buffer (refer to Section 2.8). Thus, for these parameters no data has to be copied from the message buffer

into a local variable. This solution bears the problem that a misbehaving or defective server might overwrite other data in the message buffer. But, the server will only cause its own failure, or corrupt its own data. Also, a client could transmit an ill formed message to the server, for instance containing a string that is not zero-terminated. The server could fail by trying to read the non-zero-terminated string. To avoid these buffer overruns, the DICE generated code forces zero-termination on such strings, both, when sending and when receiving. A disadvantage of referencing parameters directly in the message buffer is: if the server implementation stores these pointers in internal variables, their content will be overwritten by successive requests. Server implementers have to be aware of this fact.

Another approach to minimize copy operations has been constructed by Andreas Haeberlen as *direct stack transfer* [51] (refer to Section 2.14.1). This approach uses the stack of the client stubs generated by the target language compiler as message buffer and avoids copying the parameters from the stack into the message buffer on the same stack.

Even though techniques exist to avoid copy operations, the IDL compiler still has to produce code to copy parameters into and out of a message buffer. To make these necessary copy operations as fast as possible the IDL compiler exploits knowledge about the target platform and its specific copy operations. Processors often have a set of specialized copy instruction to speed up multimedia processing. These copy instructions are used to speed up data copy operations for marshalling and unmarshalling parameters.

4.2.3 Memory Allocation

Generated stubs have to allocate memory for variable sized data send from the server to the client. The client can avoid the memory allocation by assigning the `prealloc_client` attribute to such a variable sized parameter. The generated client code will not allocate memory, but assume that the parameter points to sufficient memory to hold the transmitted data.

Copy operations can also be reduced by utilizing the `ref` attribute with parameters. This attributes hints the IDL compiler to use an indirect part in an IPC. An indirect part describes a memory range using a pointer and size. The receiver of an indirect part has to specify into which memory range the data is to be copied. The receiver also uses an indirect part to describe this memory range.

Because this mechanism requires more work by the microkernel, it is suitable for large payloads only. The exact point of break-even depends on the implementation of the microkernel and the platform it runs on.

4.2.4 Target Language Compiler

Modern computer architectures provide speculative execution as means to optimize the utilization of a processor and thus minimize execution time. Appropriate compilers make use of this technique by placing special instructions in the instruction stream such, that probable execution paths are placed in the speculative execution path. DICE makes use

of this feature by placing hints for the target language compiler into the generated code. This also allows reducing the cache footprint of a stub, because the most common path is packed densely in the generated binary and exceptions to this common path are reached via jumps out of this densely packed code.

As mentioned above, previous IDL compilers rely on the target language compiler to perform optimization of the generated code. With some empirical measurements I found that the generated C code did not perform as well as hand-written code in assembly. Therefore, I extended the back-end for L4 version 2 by some x86 specific classes to generate assembly communication code. For specific performance numbers refer to Section 6.3.

4.3 Usability

To be able to understand the benefits of the usage of an IDL compiler in microkernel-based systems, let me first sketch the work flow of software development before the introduction of DICE. Building microkernel-based systems without an IDL compiler meant programmers spending a considerable amount of their time designing communication protocols and debugging the respective communication code. Murray et al. write in [45] that an "IDL compiler enables rapid experimentation with different interfaces." Thus, a missing IDL compiler severely limited the possibilities of the programmers.

The task of getting comfortable with all the features of the IPC system call and designing communication protocols was a high hurdle for L4 beginners. The usage of an IDL compiler lowered that hurdle, thus attracting more users to work with the system. Also, advanced features become available to beginners right away. One of the first major operating systems environments for L4–the L4Env–was only build after the introduction of the first IDL compiler.

Volkmar Uhlig provided a first implementation of an IDL compiler by implementing a back-end for L4 to Flick. This was extensively used for the SawMill project [6, 44]. It allowed a faster development of applications, because the error-prone task of writing communication code was replaced with automatic generation of communication stubs.

However, Flick had some disadvantages that had to be addressed:

- **performance**: the Flick generated code performed badly. A major requirement was to optimize the generated stubs to be faster and use less resources than existing stubs.

- **integration**: we made heavy use of existing code, such as device drivers, from other projects. Data types of this existing code should be transmitted with generated stubs. Because Flick did not allow the use of C data types, matching IDL data types had to be defined. C data types were cast to and from the respective IDL data types before and after calling generated stubs.

- **extendibility**: in a research group new ideas should be easy to implement. This includes alternative approaches to client-server communication then RPC, which

can be easily achieved by changing the communication back-end of an IDL compiler.

- **more automation**: even though Flick generated the communication code, additional code had to be written to make the stubs work, including server loops and operation identifiers. This code was almost always the same.

After analyzing Flick we came to the conclusion, the mentioned goals could only be integrated into Flick with an effort that was similar to implementing an own IDL compiler. This led to the development of the Drops IDL compiler (DICE).

A hand-written server loop for Flick stubs has to cover all special and error cases. Thus, the server loops that deviated from the default server loop were very specific and could not be reused for other scenarios. In DICE, server loops are generated. Special and error cases are handled by user-defined callbacks. This way, the generic server loop code stays the same and specific handlers can be written for the special cases. One such special case is the handling of unknown operation identifiers. Another callback can be specific to communication errors. Usually the IPC error is discarded and the IPC is repeated. Here the developer can intervene and provide an alternative reaction.

The code to be generated automatically also included the operation identifiers (opcodes). Previous work has several different mechanisms to use operation identifiers:

- `sunrpc`: opcodes are user-defined and numeric. They have to be present in the IDL file. The opcodes consisted of an interface identifier, the version of the interface, and an operation identifier.
- **MIG**: opcodes are user-defined and numeric, similar to `sunrpc` opcodes.
- **Flick**: uses user-defined opcodes. These had to be constants defined in a header file which is used by the generated stubs. The constant names had to match a predefined pattern.
- **CORBA**: opcodes are generated strings, which are aggregated from the interface name and the operation name. In [46] the usage of strings was identified as performance penalty. The authors suggested numeric opcodes to speed up dispatching.

These different approaches allowed either the developer to specify the operation identifier or the operation identifier was a generated string. DICE generates numeric opcodes. For more fine-grained control, a developer can assign opcodes to an interface or operation, which overrides the generated opcodes.

The `sunrpc` stub code generator `rpcgen` requires the version of an interface to be specified. This version is then encoded into the opcode. Thus invocations of mismatching functions should be avoided. However, sometimes a new version of an interface supports old operations, or new client libraries can handle older versions of a service. Therefore, I chose to not include the version number into the opcode, which is used for dispatching, but instead the version of an interface can be queried using specific functions. A client library can then decide to invoke the service or not.

Also, `rpcgen` has introduced extended features in recent years to overcome some deficiencies:

- C style argument passing: before, only one parameter could be handed to the communication stub. Multiple parameters had to be combined into a constructed type to be transmitted.
- Multithreading-capable communication stubs: before global variables were used in the communication stub, which hinders multi-threaded programming.
- Timeout handling in client and server stubs.

All these extended features have been available in DICE from the beginning on. For instance, timeout handling was first implemented by using a parameter in the stub functions to set the timeout value for the IPC. It turned out that the default behavior at a client is to send a request with infinite timeout and wait forever for the reply. The server waits for any requests and sends replies with timeout zero. Now, the timeout parameter is optional and is added if the timeout attribute is set. For the default behavior, optimistic stubs are generated.

4.4 Summary

In this section I showed how an IDL can provided platform independence but still platform specific optimizations can be exploited using attributes. I gave examples for using L4 specific communication mechanisms, such as indirect parts, or the transfer of flexpages. I also discussed optimization strategies for communication in generated stubs, such as parameter reordering or copy avoidance. In the following chapter I will, beside other topics, discuss the implementation of these approaches in the DICE IDL compiler.

Chapter 5

Integration

There is a wide variety of stub code generators introduced in Chapter 2. Most of them implement interesting features and ideas. This chapter will show how selected features have been integrated into the DICE IDL compiler. They range from IDL extensions, over target code generation to supporting infrastructure.

5.1 IDL Extensions

Existing interface definition languages, especially the DCE IDL, support a variety of attributes for different tasks. There exist attributes to define properties of pointers, arrays, types, etc.

Writing applications for L4 requires the specialization of parameter properties, such as identifying indirect parts or flexpages. Instead of inventing new attributes, I tried to match existing attributes from the DCE IDL to these requirements. In the following I will introduce attributes, their origin and their meaning for L4-based systems.

This section will also include newly introduced attributes if they have not been mentioned before.

5.1.1 Indirect Parts

Microkernels often provide mechanisms to transmit data *out-of-line*. That is, the message contains references to memory and not the data itself. The kernel will then copy the data from the location specified by the sender's pointer to the memory location indicated by the receiver's pointer. For the Mach microkernel this form of communication is called *out-of-line* data. The L4 microkernel calls this *indirect strings* or *indirect parts*.

The DCE IDL specification [41] contains the attribute `ref`, which is specified as:

> A reference pointer is one that is used for simple indirection. It has the following characteristics in any language that supports pointers:
>
> - A reference pointer must not have the value NULL. It can always be dereferenced.

- A reference pointer's value must not change during a call. It always points to the same referent on return from the call as it did when the call was made.
- A referent pointed to by a reference pointer must not be reached from any other name in the operation; that is, the pointer must not cause any aliasing of data within the operation.
- For **in** and **in, out** parameters, data returned from the callee is written into the existing referent specified by the reference pointer.

Because the description exactly matches to indirect parts, DICE uses the `ref` attribute to identify indirect parts. Haeberlen argues in [51] that the decision whether a parameter should be transmitted as indirect part or not should be made solely by the IDL compiler. He argues that only the IDL compiler has knowledge about the target platform and thus, only the IDL compiler can decide if `ref` can be used. We use `ref` as a platform specific attribute. This allows a user who knows the target platform and wants to benefit from it the freedom to do so. The attribute is ignored on platforms which don't support indirect parts.

5.1.2 Aliasing Types

When developing components for DROPS, existing open source code was often reused. A frequent task then is the introduction of interface specifications for this existing code. In this process, existing function declarations are often copied into an interface specification, including the argument types used.

If types cannot be transmitted in their original form, for instance, a `void*`, they are aliased with a transmittable type using the `transmit_as` attribute. The representation of the parameter in the message buffer uses the alias type. The implementation in DICE is covering more types than have originally been defined in the DCE IDL specification. The specification does not allow pointer types. However, the DCE IDL specifies that the user has to provide conversion routines for types attributed with `transmit_as`. DICE implements implicit conversion between the original and the aliased type. Providing user-defined conversion routines is discussed in Section 5.1.3.

The usage of C types in an interface specification is especially interesting if the interface specification is automatically generated from existing code. A tool can be used to extract software modules from existing software, such as drivers from the Linux kernel and generate interface specifications for them.

5.1.3 User-Defined Types

As mentioned in the previous section, the DCE IDL specification provides the `transmit_as` attribute to define an alias for the type of a parameter. According to the specification, the user has to provide conversion routines to convert the original type into the representation in the message buffer and back. With the following IDL definition:

```
[in, transmit_as(long)] my_type p1
```

The user-provided functions are:

```
long my_type_to_xmit(my_type);
my_type my_type_from_xmit(long);
void my_type_free_inst(my_type*);
void my_type_free_xmit(long*);
```

The authors of [38] suggest a similar mechanisms to extend the CORBA IDL with an attribute for user-defined types, which they dubbed `special`. The user should provide marshalling and unmarshalling routines for parameters with this attribute.

To allow the usage of user-defined types with the `transmit_as` attribute, DICE uses the conversion functions for marshalling and unmarshalling parameters with user-defined types. Known, simple types are marshalled using type casts.

5.1.4 Concurrent Data Access

Multi-threaded services will most probably operate on some common data. To synchronize access to this data the service can certainly implement or use own synchronization mechanisms. However, support to serialize the invocation of interface functions that manipulate the common data is desirable.

The Java programming language allows attributing a function with the `synchronized` keyword. This keyword indicates that execution of the function should be monitored. The monitor will only allow one thread at any time to enter the function.

One can imagine adding a similar feature to the interface definition language. However, the synchronization of a server function can highly depend on the parameters of the invoked function. Parameters can determine which part of the common data should be manipulated or if it should be manipulated at all. Thus, the synchronization strategy could be completely different for different parameter values. Another caveat is the possibility of different priorities for different threads accessing the common data. A priority inversion might be the result of a naive implementation.

Integrating a `synchronized` attribute into the interface definition language would not provide a solution for the majority of the application cases, but rather for one special case. A user-provided implementation of synchronization can provide much higher benefit.

5.1.5 Operation Identifiers

DICE automatically generates numeric operation identifiers as mentioned in Section 4.3. The operation identifier consists of a interface part, identifying the interface, and an operation part, identifying the operation in the interface. The IDL compiler can only generate operation identifiers for the scope of its knowledge.

For an interface A, which has no base interface, DICE will generate an interface identifier with the value 1. Another interface B, also without base interface, will also get an interface identifier with the value 1. Both interfaces have functions, where DICE

assigns the operation identifier 1 to the first operation, 2 to the second and so on. Now an interface C is derived from A and B. It will get an interface identifier with value 2 to distinguish the operations of C from the operation in A and B. But the dispatcher generated for interface C will have to decide whether a request with interface identifier 1 belongs to interface A or B. DICE cannot just reassign another interface number to either A or B, because, a client library for interface A compiled independently of C still has to interact with the server generated for interface C.

For cases like this, DICE generates warning messages. The user has to intervene and assign interface identifiers or operation identifiers to the interfaces or operations manually. An existing attribute that comes closest to the idea of assigning an identifier is the `uuid` attribute. In DCE IDL `uuid` is a mandatory interface attribute and is intended to uniquely identify implementations of the interface. It can be used to assign disjunctive identifiers to interfaces A and B.

I extended the scope of the `uuid` attribute to be used for operations as well. This way, further optimizations are possible. Some error and fault protocols for L4 involve the specification of special values to designated registers or positions in the message buffer. For L4 version 2 a pagefault will have the pagefault address at the position where the operation identifier is stored. This knowledge can be used to build server loops which can handle pagefaults and normal requests by using operation identifiers that cannot be used as pagefault addresses, e.g., kernel space addresses.

For L4 version 4 the position of the operation identifier will be assigned a special value in case of error messages. This way the interface specification can also be used to define a pagefault handling function. To allow the same interface specification to be used for L4 version 2 and version 4 the `uuid` attribute syntax was also extended to allow the specification of a value range. Using a pre-processor symbol, the interface definition from Section 4.2.1 looks like this now:

```
#ifdef L4API_l4v2
#define UUID_PAGEFAULT (0x0 .. 0xc0000000)
#endif

#ifdef L4API_l4v4
#define UUID_PAGEFAULT (-1)
#endif

...

[uuid(UUID_PAGEFAULT)]
void pf_handler([in, msgpos(0)] unsigned long addr,
    [in, msgpos(1)] unsigned long eip,
    [out, cap(memory)] l4_fpage_t page);
```

5.1.6 Array Size

Arrays are represented in the target language C/C++ as continuous bytes in memory. The size of a fixed sized array is known at compile time. On the other hand, a generated

stub cannot determine the size of a variable sized array from the parameter it receives. The DCE IDL provides the attributes `size_is`, `length_is`, and `max_is` for arrays. These attributes take another parameter as argument. Thus, the IDL compiler can determine the space required.

A server has to allocate memory for a message before it actually receives the message. Also, because the server may receive different messages with differently sized indirect parts, it has to preallocate for the biggest possible indirect part.

I also enabled these attributes in DICE. However, they indicate a much severe problem for microkernel-based systems: The usage of variable sized parameters whose size is only known at run-time. A variable sized `out` parameter implies that: a) the client has to provide a message buffer of unknown size and b) the sever stub has to be able to free dynamically allocated memory of a variable sized array.

I solved the first problem by letting the user specify a maximum size for the variable sized parameter using the `max_is` attribute. If none is specified, DICE uses a heuristic to determine the maximum size. The heuristic is based on empirical data for various array sizes and DICE warns about the usage of the built-in heuristic. The client stub allocates a message buffer that can hold a maximum sized parameter. The following example shows that the variable sized `array` has a maximum size of 255 elements and contains `size` elements at run-time.

```
void send_array([in] int size,
    [in, size_is(size), max_is(255)]
        int array[]);
```

If a server dynamically allocates memory for a variable sized `out` parameter, the content of this memory is usually copied to the message buffer and then the memory is freed. To avoid copy operations, such a parameter can also be specified as out-of-line data (see Section 4.1). The memory holding the parameter has to be available during transmission to the client, which is an atomic operation. This leads to the second problem: memory can only be freed after the next request has been received, i.e., it is stale until the next request arrives. The next request is most likely for a different operation and parameters and, thus, the server has no longer a name for the previously allocated memory. Therefore, the server has to store the address of the dynamically allocated memory and free it after the next request is received.

This is done by using the DICE support library. It allows storing pointers in the server local environment. These pointers are not changed by the next request. Another function iterates over the stored pointers and frees the associated memory. This function is invoked after receiving the next request from the generated code.

5.2 Target Code Generation

The following section includes different techniques applied to the target code generation of DICE.

5.2.1 Factory Concept

To be able to integrate multiple target platforms into DICE, we need a target code creation process that is independent of the target platform. In [43] the authors describe a mechanism that helps creating back-ends for specific target platforms using *abstract factories*. The indicators for the applicability of an abstract factory are a) independence of the whole system (DICE) from the way its back-ends are created, b) the system should be configured with one of multiple back-ends (only one target platform is used at one time), c) a family of related back-end objects is designed to be used together (only classes for one target platform are created), and d) the intent to provide a class library of back-ends, of which only their interfaces, not their implementation, are revealed. All these indicators apply. Therefore, the usage of an abstract factory is appropriate.

Class Factory To create the classes of the different back-ends, I implemented a class factory, which provides a method for each class in the back-end. Whenever an instance of a class is needed, the respective method of the class factory is called and returns a reference to a newly created object of the requested class. When DICE finishes parsing the IDL file and starts to create the back-end for the target platform it first creates an instance of the appropriate class factory. The class factory itself is a singleton, that is, there exists only one instance of it.

Since concrete back-end classes cannot be initialized generically, a second class-specific method–`CreateBackEnd`–has to be called to finish the creation process. For instance, the `CBEFunction` class is initialized with a respective `CFEOperation`, whereas a `CBEClass` is initialized with the respective `CFEInterface` class.

To add a new back-end, the classes in the back-end have to be overloaded. To actually use these classes, a new class factory for this back-end has to be derived from the base class factory. The new class factory has to be instantiated when the new back-end is selected.

This design can be extended to dynamically load alternative back-ends. The class factories for the different back-ends have to register at a class-factory factory, which then creates the required class factory as soon as a specific back-end is selected. The mapping from user-selected option to back-end can be done using maps that match an option string to the respective instance of the class factory.

Name Factory There exists another factory in DICE: the name factory. It is used to generate the names of types, functions, classes and the like. Whenever an identifier in the target language is used, the name factory is queried to provide an appropriate string. This is used when platform specific types are used and the strings identifying these types differ: `l4_fpage_t` versus `L4_Fpage_t`. If the user selects the usage of a strict CORBA language mapping, CORBA compliant type names are generated instead of C type names.

The name factory creates different names for client and server functions. This is useful if an application implements an interface and uses the client library of the same

interface, for instance as a proxy. Also, descriptive names of client and server functions prevent accidental use of the wrong generated functions.

The name factory can also be used to integrate user-provided language mappings. By dynamically loading back-ends, an alternative name factory can be used to generate different type or function names. A name factory provides a flexible and extensible way for different naming schemes. As discussed in Section 2.16 a static language mapping is not the best solution, especially if the generated stubs have to be integrated with existing code.

5.2.2 Message Buffer Layout

One representation of a message buffer is a byte array. Marshalling is done by copying parameter values to a specific offset in the message buffer. This approach has some limitations. The following code schematically marshals two short integer parameters and one word parameter into the message buffer. The message buffer is cast to the type of the parameter and then the parameter is assigned. The message buffer is then used in the IPC invocation.

```
1    unsigned char msg_buf[8]; /* 8 bytes size */
     /* marshal short int parameter */
3    *(short int*)(& msg_buf[0]) = param_s1;
     *(short int*)(& msg_buf[2]) = param_s2;
5    *(long *)(& msg_buf[4]) = param_w;
     /* send message */
7    l4_call (server, msg_buf,
         *(unsigned long*)(& msg_buf[0]),
9        *(unsigned long*)(& msg_buf[4]), ...);
```

The target language compiler sees five accesses to the byte array: Writes at index zero (line 3), index two (line 4), and index 4 (line 5) and reads at index zero (line 8) and index four (line 9). Some target compilers try to optimize cache locality and collocate the two accesses at index zero and index four. So the sequence is now:

```
write at index 0 with length 2
read at index 0 with length 4
write at index 2 with length 2
write at index 4 with length 4
read at index 4 with length 4
```

This is evidently incorrect. To overcome it I replaced the byte array message buffer with a union of constructed types. Here, the parameters for one direction are combined into a constructed type. The two constructed types for the two directions are integrated into a union, which allows saving memory, because the two constructed types overlay. The union additionally contains a constructed type with an array of word sized elements. The above example looks like this now:

```
1    union {
         struct {
3                short int param_s1;
```

```
                short int param_s2;
5               long param_w;
        } _in;
7       struct {} _out;
        struct {
9               unsigned long _word[2];
        } _words;
11  } msg_buf; /* still only 8 bytes size */
    /* marshal short int parameters */
13  msg_buf._in.param_s1 = param_s1;
    msg_buf._in.param_s2 = param_s2;
15  msg_buf._in.param_w = param_w;
    /* send message */
17  l4_call (server, (unsigned long*)&msg_buf,
        msg_buf._words._word[0],
19      msg_buf._words._word[1], ...);
```

The code is a little longer in C, but has the same memory consumption of the above example, the same code size in the binary, and is semantically not ambiguous.

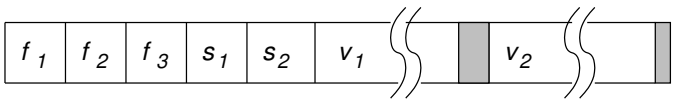

Figure 5.1: Message buffer layout with variable sized members.

Variable sized parameters still have to be unmarshalled from a byte array. This is necessary, because when receiving a variable sized parameter we don't know its actual size in advance. Variable sized members are represented in the message buffer by their size followed by the actual data. The position of the successive member in the message buffer is: $pos(v_2) = roundup(pos(v_1) + s_1, sizeof(word))$.

As pictured in Figure 5.1, all variable sized parameters ($v_{1/2}$) are placed into a byte array together with their actual sizes ($s_{1/2}$).

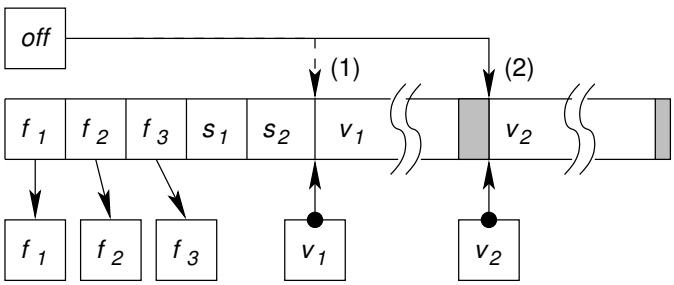

Figure 5.2: Unmarshalling with variable sized members.

Figure 5.2 illustrates unmarshalling of this message buffer, all fixed sized members $f_{1..3}$ are copied into local variables. Then the local offset variable is initialized with the start offset of the first variable sized member of the message buffer (1). When optimizing access to the message buffer, a local variable for the member will only point to the start of the variable sized member in the message buffer. No copy operation will occur. The offset variable is then incremented with the size of the variable sized member s_1 and rounded to the next word aligned address (2). Using a word aligned address to store the next size variable and variable sized member, can save computation time when using optimized copy routines.

5.2.3 Tracing

In software development bugs can appear everywhere. Thus debugging is a time consuming and major part of it. One approach to debug code is to generate traces, e.g., by printing current state. To allow the debugging of generated code and to trace the communication between clients and servers, DICE can generate tracing code in the generated stubs. For this purpose, DICE provides a default tracing facility that simply prints the state of client and server stubs. This tracing facility can be replaced dynamically, that is, with a dynamically loaded library. This way arbitrary code can be used to generate other traces.

The following relevant trace points have been identified [23]:

- before and after a call
- at server initialization
- before starting the server loop
- before and after invoking the dispatch function
- before and after the invocation of reply function
- before and after the reply and wait function
- before and after invoking the component function
- before and after marshalling and unmarshalling a message

Alternatively, tracing code can be injected into the generated stubs using Aspect Oriented Programming. But Aspect compilers for the C target language did not exist or were not powerful enough at the time of this writing.

5.2.4 Test Suite Generation

DICE had the feature to automatically generate test suites for the parsed IDL files. The generated code assumed a run-time environment in which threads could be started and

terminated as well as output could be generated. Also service registration and discovery had to exist.

The generated code would then call each function with random values. The transferred values were also stored in memory shared between client and server so the receiver could compare the received values with the stored values.

This approach had some major downsides: Firstly, DICE became overly complex. A detailed analysis of the complexity of the code generating the test suite can be found in Section 6.4.1. Secondly, using the IDL compiler to generate its own test suite and thus to verify itself is problematic. Also, testing for a new platform required that the test-suite generating code had to accommodate the specifics of the new platform. This basically doubles the effort to integrate a new platform, as the communication code *and* the test suite had to be implemented for the new platform.

Therefore, I extracted the test suite into an own software package. The test suite then uses DICE to generate the communication stubs. The framework for the test was not generated any longer. This allowed testing more sophisticated scenarios, especially rare corner cases. Also, tests could be called repeatedly now to detect memory leaks. Compared to the code extracted from the DICE IDL compiler, the hand-written test suite was rather small.

5.3 Infrastructure Integration

Generated code often has to use existing infrastructure to provide a service. This has been motivated in Section 3.2 with the example of a thread library.

5.3.1 Automatic Service Lookup

A reoccurring functionality that has to be implemented in microkernel-based operating systems is the announcement and lookup of a service. Because the used code is mostly the same, it is apparent that this code should be generated by an IDL compiler. In CORBA the naming of services, their announcement, and lookup are an integral part.

The generated code has to utilize functionality that is, similarly to the thread creation mentioned before, external. The DICE library provides functions to announce a service and to look it up. An attribute associated with an interface indicates whether the generated code should contain the statements to use the naming service.

Still, the name of the service has to be defined. The DCE IDL `endpoint` attribute could be used for that: L4 services use a new "l4::names" protocol and the socket backend uses the TCP/IP protocol.

5.3.2 Resource Accounting

A difficult problem for communication is resource accounting, that is, which party provides the resources for a service. Usually, the client has to provide the resources to get its requested service. In a microkernel-based system this includes basic resources

such as memory and computation time. To delegate resources from client to server, the generated code has to integrate with a resource accounting framework.

Currently existing servers provide the resources they need to fulfill a service themselves. This implies that a client may not be charged for the used resources. This includes resources which are required to be able to invoke a service and resources which are used during a request.

Ideally the resources used during a request should be charged to the client invoking the service. The client could provide the resources in the call, e.g., by supplying capabilities to the resources [88, 36] or economic resource accounting [76, 77, 78].

A naive approach, to provide memory from a client to the server, is to map the required amount of memory to the server with each request. The memory may already contain the data for the request and a stack to process the request. Providing memory for heap memory allocations is more complicated, since memory allocation functions have to get their memory from the user-provided memory pages.

This approach has some open issues: In a scenario where the invoked service has to use other services, it has to propagate the resources or a subset thereof to the third party. Another open question is the reaction on resource revocation by a client or resource shortage, which means, the provided resources are not enough. One possible solutions is to use a trusted third party, which manages the resources and is used by client and server [87]. Also, the mapping of pages is a costly operation.

I already discussed the propagation of computation time from the client to the server in Section 4.1.

5.3.3 Resource Reservation

In real-time or quality of service scenarios, the used resources have to be reserved in advance, so that no resource contention degrades the quality of a service or causes deadline misses. For these scenarios we developed a notation to specify *provided quality of service* and *required resources*. To reserve resources the generated code has to integrate with an existing resource reservation framework.

Within the Comquad project, we developed means to describe resources associated with components and consequently services [5, 19]. Resources used by a component are *required* to *provide* a certain quality of service. The higher level description of these resource mappings can be directly translated into resource descriptions associated with an interface in an ACF.

If such a resource description is specified, the generated code for the server is extended by a resource reservation function. The server then uses this function to interact with the resource reservation framework and reserve resources required for its service.

5.4 Real-Time Communication

Communication with real-time applications has some major implications. One is that resources for the communication have to be available when needed. The issue of resources

and their availability has been discussed in Section 5.3.3.

Another implication of real-time communication is that the delivery of a message has to be bound in time. When analyzing possible communication scenarios with real-time applications, three interesting scenarios appear.

1. communication between two real-time applications,
2. communication from a non-real-time application to a real-time application, and
3. communication from a real-time application to a non-real-time application.

In the first two scenarios, the actual service is assumed to adhere to timing constraints. The timely execution of a request then depends mainly on the time needed to send and dispatch a message. For dispatching the message an upper bound can be given. Stubs generated by DICE may depend on dynamic memory allocation. The developer of a real-time service can use allocation algorithms with bounded execution times for memory allocation or avoid the usage of dynamically allocated memory. Pointers to these functions can be stored in the environment, which are used to allocate and free memory in the generated stubs. Thus, these scenarios can be predictable in their execution time.

The third scenario, however, cannot really be predicted. Because we don't know the behavior of the non-real-time application, the sending real-time application may block forever waiting for the receiver to accept the message. In [83] we discussed this problem in depth and propose a buffer component, which buffers the messages of the real-time component. The buffer component decouples the real-time applications from the non-real-time applications.

The buffer component receives the messages of the real-time application and places them into a queue. Therefore, the buffer component has a receiver thread for each connected real-time application. A sender thread will take messages from the queue and propagate them to the non-real-time applications. A real-time application can no longer be influenced directly by the non-real-time application. If the non-real-time application is not ready to receive messages, the buffer component replaces messages in the queue. It can implement an arbitrary predictable replacement strategy. This will allow the real-time application to deliver messages, which can be dropped, but it will not block to wait for the buffer component.

5.5 Summary

In this section I showed how existing designs for interface definition languages and their compilers can be integrated into an IDL compiler for microkernel-based operating systems. I also discussed new extensions to the IDL to allow support of platform specific features. The compiler is flexible to integrate support for different programming environments. I gave explicit examples for these different environments, such as resource accounting or resource reservation. I also discussed how real-time applications can use generated stubs and still maintain their timing guarantees.

Chapter 6

Evaluation

This chapter will discuss performance measurements and generated code size, as well as, compare the complexity of the stub code generators used.

The experimental results will show performance of the generated code in a detailed analysis of different communication forms in micro-benchmarks. Real-life application benchmarks will show the overall overhead introduced by the IDL compiler.

If not specified otherwise, I use cycles when comparing performance of code. There are advantages and disadvantages to this method. One advantage is that computing cycles can show fine-grained differences on fast computers. Being able to compare the same code on different machines with a simple metric to distinguish performance penalties due to the architecture, is another one. On the other hand, even though the same code can use more cycles on newer hardware, it can be faster because of higher clock rates.

6.1 Analyze Method Invocations

Because the measurement of complex setups involves side effects and generates imprecise results, I first analyzed different setups to gain an understanding of exchanged messages. Using this knowledge I build a benchmark, which did not involve the complex setups, but measures only the used messages. This allows me to give precise results for the generated stubs without the need of investigating side effects of applications running in parallel.

First I trace the number of differently sized IPC messages in the complex setups with an extension to the Fiasco kernel. Then I instrumented the generated IDL stubs for all applications participating in a particular setup using the mechanism mentioned in Section 5.2.3. This allowed me to identify the number of invocations of single IDL methods. Data was logged periodically using the logging service. Because the communication with the logging service involves IPC, the instrumented stubs of the logging service influenced the measurements.

The gathered information allowed me to generate a representation of all the IDL methods involved in different setups. I am able to weight single methods based on the

number of times they have been invoked and their size.

The instrumented scenarios were:

- L4Linux with L4 console
- L4Linux with L4 console and ORe
- L4Linux (3x) with L4 console and ORe (aka. Netfilter scenario)
- L4Linux with DOpE
- Verner
- jtop
- BLAC

The results presented below will show most messages occur at startup of the system. For some interactive scenarios I split the measurements in startup phase and run-time phase.

6.1.1 Scenario 1: L4Linux with L4 console

This scenario starts an instance of a para-virtualized Linux on top of L4. L4Env services are used to access virtualized hardware resources. These services include the `log` service to multiplex debug output of different applications, the dataspace manager service (`dm`) to virtualize access to memory, the L4 console service (`con_vc`) to multiplex access to the screen, the naming service `names` to locate tasks, and the task service `ts` to create and delete tasks.

After booting, multiple tasks are started, which reflects in the invocations of `rmgr` service's `get_task` and `task_new` methods, as well as `ts`'s `allocate` and `create` methods. The memory of the tasks has to be mapped as well. This shows in the number of invocations of the `dm` service's `fault` and `map` methods.

interface::method	invocations
log::outstring	7334
dm_generic::map	12902
dm_generic::fault	14578
dm_generic::close	22
dm_generic::share	6
dm_generic::transfer	46
dm_mem::open	115
dm_mem::size	16
dm_mem::resize	28
dm_mem::physaddr	4

continued on next page...

interface::method	invocations
dm_phys::pagesize	7
dm_phys::poolsize	73
names::register	9
names::register_thread	43
names::query_name	62
rmgr::init_ping	22
rmgr::get_task	316
rmgr::task_new	24
rmgr::get_irq	16
rmgr::free_fpage	1
rmgr::get_page0	1
rmgr::get_task_id	10
rmgr::set_task_id	24
ts::allocate	19
ts::create	33
ts::free	14
ts::kill	28
ts::owner	1
con_vc::smode	1
con_vc::gmode	1
con_vc::get_rgb	1
con_vc::setfb	1
con_vc::direct_update	1953

Table 6.1: Number of invocations for L4Linux with L4 console counted with instrumented stubs

interface::method	calculated size in word	
	to server	from server
log::outstring	5 (I[1]: 81)	1
dm_generic::map	7	6 (F[2])
dm_generic::fault	3	6 (F)
dm_generic::close	2	2
dm_generic::share	4	2
dm_generic::transfer	3	2
dm_mem::open	13	4
dm_mem::size	2	3
dm_mem::resize	3	2
dm_mem::physaddr	4	4

continued on next page...

[1]Indirect string used, size in bytes
[2]Flexpage used

interface::method	calculated size in word	
	to server	from server
dm_phys::pagesize	5	3
dm_phys::poolsize	2	4
names::register	10	2
names::register_thread	11	2
names::query_name	10	3
rmgr::init_ping	2	2
rmgr::get_task	2	2
rmgr::task_new	9	3
rmgr::get_irq	2	2
rmgr::free_fpage	2	2
rmgr::get_page0	1	5 (F)
rmgr::get_task_id	258	3
rmgr::set_task_id	259	2
ts::allocate	1	3
ts::create	267	3
ts::free	2	2
ts::kill	3	2
ts::owner	3	2
con_vc::smode	3	2
con_vc::gmode	1	6
con_vc::get_rgb	1	8
con_vc::setfb	3	2
con_vc::direct_update	3	2

Table 6.2: Size of IDL method calls for L4Linux with L4 console counted with instrumented stubs

In Figure 6.1 I plotted the number of IPC messages that make up 95% of the total number of IPCs in this scenario. As soon as the scenario starts, the applications establish their address spaces. The manipulation of the address space of an application is done using the `map` and `fault` methods, which are invoked most. The method with the third most invocations is the `outstring` method, followed by the `update` method for the console.

To compare not just the number of invocations, but also the amount of data transferred with these methods, I weighted the number of invocations with the size of the respective method (refer Table 6.2). The resulting chart is shown in Figure 6.2. As one can see, the methods `fault` and `map` moved to places two and three. The message causing the most data transfer is `outstring`. It makes up almost 50% of the data volume. However, this chart shows only the size of the `fault` and `map` messages. These messages initiate much more work in the kernel than just transferring the data. The kernel has to establish a memory mapping initiated by these messages.

An important result of this analysis is, that a handful of messages are the source of

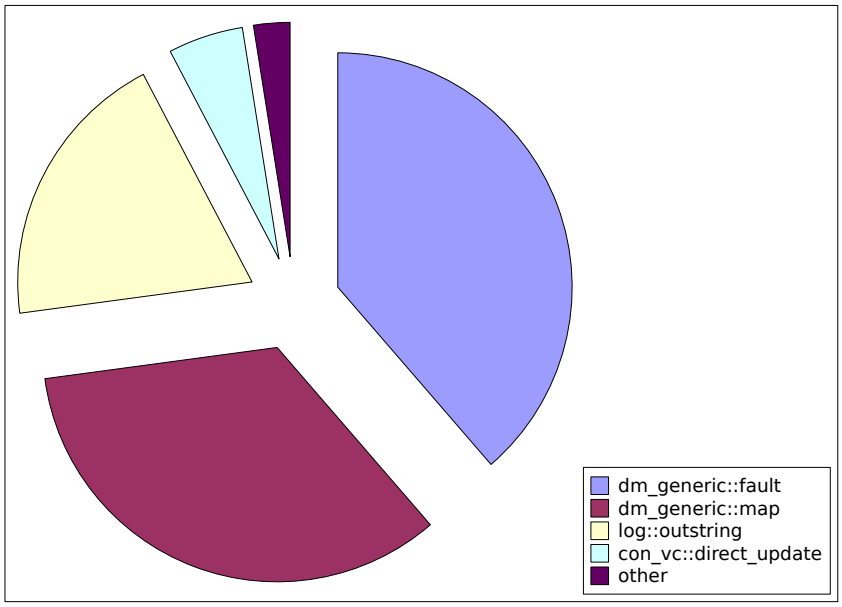

Figure 6.1: Portions of message invocations in comparison to overall messages for L4Linux with L4 console scenario.

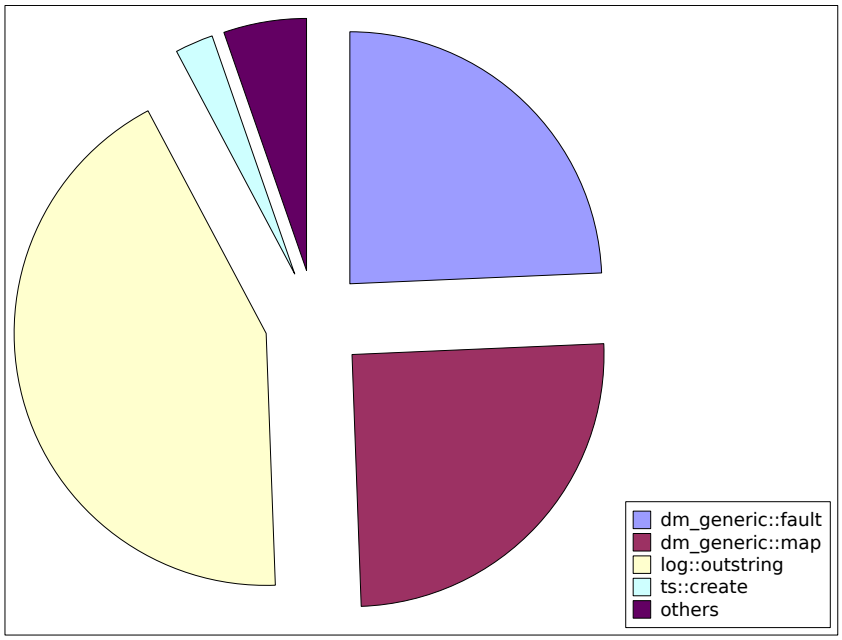

Figure 6.2: Weighted message invocations in comparison to overall data volume for L4Linux with L4 console scenario.

95% of the data volume transferred in this scenario.

Once this scenario has started, the number of IPC invocations decreases. Then, the most used service is `con_vc` to refresh the console output of L4Linux. Also, the `rmgr` and `dm` services are invoked whenever a new Linux task is started.

In this scenario, the messages that can be send using register IPC (short IPC) are message with a size of 2 or less machine words. 10,367 of the generated messages fall into this category, which is approximately 14% of all the messages sent with generated code.

The majority of generated messages without flexpages or indirect strings (63%) is less than 32 words in size. Another 36% make up the messages containing a flexpage and less than 0.1% of the messages are larger than 32 words.

The aforementioned kernel extension counted a total of 85,726 register IPC invocations (refer to Table 6.3). The difference to 10,367 messages can be explained with pagefault messages generated by the microkernel and messages sent using hand-written IPC code.

Whenever a L4Linux user task is invoking a system call, the Fiasco kernel will generate an exception message (similar to the pagefault protocol) and send it to the L4Linux server. This server handles the exception (performs the system call) and replies with an appropriate reply IPC. An exception message captures the volatile state of the architecture and is 16, 20, or 23 words in size for x86, ARM, and AMD64 respectively. Because I measured on an x86 architecture the exception messages are 16 words in size. In this scenario, the kernel extension did not count any exception IPCs.

IPC Sizes in words		generated stubs	total	difference
lower bound	upper bound			
0	2	10367	85726	75359
3	3	16807	15294	-1513
4	4	202	5691	5489
5	5	8	127	119
6	6	27481	25214	-2267
7	8	12903	11808	-1095
9	16	255	6697	6442
17	24	0	67	67
25	32	7334	0	-7334
33	64	0	0	0
65	96	0	3	3
97	128	0	0	0
129	192	0	0	0
193	256	0	0	0
257	512	67	327	260
513	1024	0	0	0
1025	2048	0	0	0

continued on next page...

IPC Sizes in words		generated stubs	total	difference
lower bound	upper bound			
2049	4096	0	0	0
4097	8192	0	0	0

Table 6.3: Number of IPCs sorted by size for L4Linux with L4 console counted by kernel extension

In Table 6.3 I sorted the number of IPC invocations by message size. The column *generated stubs* repeats the number of messages from Table 6.2. The column *total* shows the number of invocations counted with the kernel extension. This column includes messages initiated by generated stubs as well as messages generated by the kernel or hand-written code.

The message sizes of the generated stubs are maximum sizes. The actual transmitted size can be smaller. For instance, a simple IDL string without annotations is assumed to have a maximum size of 512 bytes. When actually transmitted, the string might contain only 32 or 80 bytes. The *difference* column shows the difference between these two numbers. Negative values express more messages derived from the instrumented stubs. Positive numbers indicate more IPC counted by the kernel extension.

6.1.2 Scenario 2: L4Linux with L4 console and ORe network

Additionally to the setup described in the previous section, this setup virtualizes the access to the network device using the `ore` service.

interface::method	invocations	
	Scenario 1	Scenario 2
log::outstring	7334	17765
dm_generic::map	12902	16747
dm_generic::fault	14578	14809
dm_generic::close	22	28
dm_generic::share	6	7
dm_generic::transfer	46	55
dm_mem::open	115	137
dm_mem::size	16	12
dm_mem::resize	28	26
dm_mem::physaddr	4	8
dm_phys::pagesize	7	7
dm_phys::poolsize	73	179
names::register	9	10
names::register_thread	43	50
names::query_name	62	111
rmgr::init_ping	22	25

continued on next page...

interface::method	invocations	
	Scenario 1	Scenario 2
rmgr::get_task	316	316
rmgr::task_new	24	24
rmgr::get_irq	16	16
rmgr::free_fpage	1	1
rmgr::get_page0	1	1
rmgr::get_task_id	10	11
rmgr::set_task_id	24	24
ts::allocate	19	19
ts::create	33	33
ts::free	14	14
ts::kill	28	28
ts::owner	1	1
con_vc::smode	1	1
con_vc::gmode	1	1
con_vc::get_rgb	1	1
con_vc::setfb	1	1
con_vc::direct_update	1953	2371
ore_rxtx::send	–	17755
ore_rxtx::recv	–	17914
ore_notify::rx_notify	–	17759

Table 6.4: Number of invocations for L4Linux with L4 console and ORe counted by instrumented stubs

In Table 6.4 I compare the number of invocations to the previous scenario (Scenario 1). It shows that a common subset of messages is needed to boot an L4Linux server. Additionally, we now have a tremendous increase in the share of messages larger than 32 bytes – 18% compared to less than 0.1%. This is due to the network device server, which exchanges data with its clients using indirect string IPC.

6.1.3 Scenario 3: Netfilter

The Netfilter setup uses three L4Linux instances, which all access the `ore` network device service. This setup emulates a networking filter the TU Dresden operating systems group build for the Mikro-Sina project to securely filter network traffic [54]. One L4Linux instance has access to the Internet. The second L4Linux instance has access to the local network. A bridge application connects the two instances and implements a filter that denies confidential data to be leaked from the intranet to the Internet. It also prevents unauthorized traffic from the Internet into the intranet. Using the filter setup prevents the internal L4Linux to be infiltrated, even if the external L4Linux is compromised. I used a third L4Linux instance to emulate the filter application.

In this setup I generated network traffic from the L4Linux instance connected to the intranet. This traffic was routed by ORe to the filtering L4Linux instance, which sends the allowed traffic on to the L4Linux instance with Internet access. In Table 6.5 I again compare the number of invocations from scenario 1 with this scenario.

interface::method	invocations	
	Scenario 1	Scenario 3
log::outstring	7334	58634
dm_generic::map	12902	35871
dm_generic::fault	14578	43900
dm_generic::close	22	72
dm_generic::share	6	21
dm_generic::transfer	46	89
dm_mem::open	115	260
dm_mem::size	16	14
dm_mem::resize	28	76
dm_mem::physaddr	4	16
dm_phys::pagesize	7	25
dm_phys::poolsize	73	615
names::register	9	10
names::register_thread	43	73
names::query_name	62	126
rmgr::init_ping	22	27
rmgr::get_task	316	347
rmgr::task_new	24	69
rmgr::get_irq	16	16
rmgr::free_fpage	1	1
rmgr::get_page0	1	1
rmgr::get_task_id	10	12
rmgr::set_task_id	24	69
ts::allocate	19	45
ts::create	33	69
ts::free	14	27
ts::kill	28	48
ts::owner	1	3
con_vc::smode	1	3
con_vc::gmode	1	3
con_vc::get_rgb	1	3
con_vc::setfb	1	3
con_vc::direct_update	1953	16235

continued on next page...

interface::method	invocations	
	Scenario 1	Scenario 3
ore_rxtx::send	–	51508
ore_rxtx::recv	–	52039
ore_notify::rx_notify	–	51533

Table 6.5: Number of invocations for Netfilter startup counted by instrumented stubs

The output of the measurements generates many IPCs (log::outstring) as does the periodic update of the console window (con_vc::direct_update). Once the initial startup is complete any other IPC activity decreases significantly (see Table 6.6). There is communication with the dataspace manager whenever a new L4Linux user task is started. And the console application periodically queries the pool-size at the dataspace manager to displays the amount of used memory.

The number of IPCs to filter a network stream is relatively low compared to the number of IPCs that were exchanged for startup. The loader loads binaries and libraries over the network device that is now managed by ORe. Because three L4Linux instances are started, the L4Linux binary and dependent libraries have to be loaded three times. These binaries and libraries caused more network traffic than the data that was filtered in this scenario.

interface::method	invocations
log::outstring	97394
dm_generic::map	8
dm_generic::fault	18
dm_generic::transfer	3
dm_mem::open	10
dm_phys::poolsize	2564
names::register_thread	20
names::query_name	18
rmgr::get_task	161
rmgr::task_new	165
rmgr::get_page0	1
rmgr::set_task_id	165
ts::allocate	85
ts::create	165
ts::free	80
ts::kill	160

continued on next page...

interface::method	invocations
con_vc::direct_update	399103
ore_rxtx::send	494
ore_rxtx::recv	606
ore_notify::rx_notify	139

Table 6.6: Number of invocations for Netfilter running counted by instrumented stubs

6.1.4 Scenario 4: L4Linux with DOpE

In this scenario the console service is replaced with the DOpE window manager. L4Linux now opens a window and displays its output in this window. The main difference to the previous setups is, that L4Linux no longer has to update the screen periodically. Updates show up immediately. The periodic invocation of con_vc::direct_update is replaced by calls to dope::exec_cmd and dope::exec_req.

interface::method	invocations
log::outstring	6802
dm_generic::map	11434
dm_generic::fault	14848
dm_generic::close	23
dm_generic::share	6
dm_generic::check_rights	1
dm_generic::transfer	46
dm_mem::open	142
dm_mem::size	23
dm_mem::resize	25
dm_mem::physaddr	5
dm_phys::pagesize	7
dm_phys::poolsize	2
names::register	13
names::register_thread	69
names::query_name	70
rmgr::init_ping	22
rmgr::get_task	316
rmgr::task_new	24
rmgr::get_irq	16
rmgr::free_fpage	1
rmgr::get_task_id	10
rmgr::set_task_id	24
ts::allocate	19

continued on next page...

interface::method	invocations
ts::create	33
ts::free	14
ts::kill	28
ts::owner	1
dope::init_app	1
dope::exec_cmd	3152
dope::exec_req	1

Table 6.7: Number of invocations for L4Linux + DOpE counted by instrumented stubs

Because DOpE has been designed to work with both, L4 IPC and Linux sockets, its interface does not exploit optimization features available for L4. Comparing the size of dope::exec_cmd (259 and 2 words) to the size of con_vc::direct_update (3 and 2 words) underlines this statement. The latter is designed to update a console screen with an interface designed for L4. The former is multiplexing many commands over a simple interface, one being the update command.

6.1.5 Scenario 5: Verner – video player

This scenario is a video player application [85], which consists of five components, each running in separate tasks. Figure 6.3 shows that the first component demultiplexes a video into audio and video streams. These streams are than decoded by audio and video decoders respectively. The synchronizing component then plays the audio streams and displays the video in sync. The data streams between the components are exchanged using shared memory. The fifth component is the controller, which manages the other four components and provides the graphical user interface (GUI) to the user.

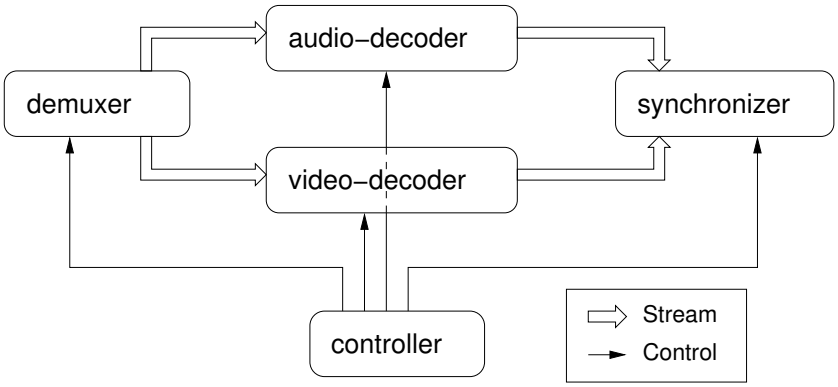

Figure 6.3: Schema of Verner video player.

Once the *Start* button of the player GUI is pressed, the controller sets up the shared memory regions and invokes the start method of each component. The components will start consuming packets from the shared memory regions. The setup and management of the shared memory regions is handled by the DROPS Streaming Interface (DSI) library [71]. This library communicates using hand-written IPC.

In this scenario I use a 40 MB video file, which is loaded at startup into the address space of the demultiplexing component. This component copies single frames into the shared memory regions after demultiplexing the input file.

interface::method	invocations
log::outstring	2580
dm_generic::map	66569
dm_generic::fault	48539
dm_generic::close	43
dm_generic::transfer	379
dm_mem::open	1063
dm_mem::size	121
dm_mem::physaddr	184
dm_phys::pagesize	35643
names::register	13
names::register_thread	69
names::query_name	70
rmgr::init_ping	27
rmgr::get_task	64
rmgr::task_new	7
rmgr::get_irq	16
rmgr::get_task_id	12
rmgr::set_task_id	7
ts::allocate	7
ts::create	6
ts::owner	6
dope::init_app	1
dope::exec_cmd	151

Table 6.8: Number of invocations for Verner startup counted by instrumented stubs

The high numbers of `dm_phys::pagesize` calls in Table 6.8 are generated by the loader when it is paging an application. It first checks if it can send a page as 4MB page. This check is done on every pagefault. Because the loader loads five binaries in this scenario, it is the initial pager for all five applications. This amounts in more than 35,000 `pagesize` calls.

For this scenario I did two consecutive runs. One measured the invocations of the

generated stubs. The other measured the size and number of IPCs. In both runs I marked the transition between startup and "running" by hand. This and the fact that both runs took a different absolute length of time are reasons for the variance in both measurements.

The invocations of `dm_phys::pagesize` are followed by calls to `dm_generic::map`.

interface::method	invocations
log::outstring	2056
dm_generic::map	14694
dm_generic::fault	17249
dm_generic::close	4
dm_generic::share	98
dm_generic::check_rights	102
dm_generic::transfer	477
dm_mem::open	11537
dm_mem::size	11413
dm_mem::resize	179
dm_phys::pagesize	3943
names::register	1
names::register_thread	27
names::query_name	10
verner::connect	4
verner::start	8
verner::changeQAP	1
verner::getPosition	428
verner::setVolume	5
verner::setPlayback	2
verner::setFxPugin	4
dope::init_app	3
dope::exec_cmd	1189
dope::exec_req	5

Table 6.9: Number of invocations for Verner running counted by instrumented stubs

6.1.6 Scenario 6: JTop – a DOpE application

JTop is a performance monitor, which makes high use of the DOpE window system. It repaints its window every second. It is a perfect scenario to demonstrate the communication pattern of applications using the DOpE windowing system.

interface::method	invocations
log::outstring	1767
continued on next page...	

interface::method	invocations
dm_generic::map	245
dm_generic::fault	1169
dm_generic::close	3
dm_generic::share	2
dm_generic::check_rights	2
dm_generic::transfer	44
dm_mem::open	156
dm_mem::size	70
dm_phys::pagesize	34
names::register	8
names::register_thread	50
names::query_name	77
names::query_id	1329
rmgr::init_ping	16
rmgr::get_task	64
rmgr::get_irq	8
rmgr::get_task_id	7
dope::init_app	2
dope::exec_cmd	9668
dope::exec_req	1

Table 6.10: Number of invocations for JTop counted by instrumented stubs

As shown, the method used most frequently is `dope::exec_cmd`, which tunnels all the display commands for the windowing system. Due to the generic design of the DOpE interface, it is using a maximum sized buffer to transmit the commands. Using L4 specific attributes to set the actual size of the buffer at run-time would dramatically reduce the message sizes and consequently improve performance.

6.1.7 Scenario 7: BLAC

The BLAC scenario [89] consists of an L4Linux with hybrid Linux-L4 applications and L4 services providing a secured transaction mechanism to the hybrid Linux applications. The L4Linux server relies on some L4 services and some of the L4 services rely on functionality provided by Linux applications. This scenario has complex timing constraints during startup. The log and dataspace manager interface have not been instrumented in this scenario, since they interfere too much with these timing constraints.

interface::method	invocations
rmgr::init_ping	50
rmgr::get_task	734
continued on next page...	

interface::method	invocations
rmgr::task_new	368
rmgr::get_irq	27
rmgr::free_fpage	1
rmgr::get_task_id	27
rmgr::set_task_id	370
ts::allocate	215
ts::create	377
ts::free	178
ts::kill	339
ts::owner	11
ts::exit	1
dopeapp::event	1197
dope::init_app	2
dope::exec_cmd	116626
dope::exec_req	1210
names::register	21
names::register_thread	115
names::query_name	5275
names::query_id	25
nitevent::event	247
overlay::get_screen_info	1
overlay::open_screen	1
overlay::map_screen	1
overlay::refresh_screen	1092
overlay::input_listener	1
overlay::window_listener	1
overlay::create_window	17
overlay::destroy_window	7
overlay::open_window	18
overlay::place_window	17
overlay::stack_window	78
overlay::set_background	1
ore_rxtx::send	284876
ore_rxtx::recv	288612
ore_notify::rx_notify	271344

Table 6.11: Number of invocations for BLAC counted by instrumented stubs

6.1.8 Summary

Using the above measurements, I generated a list of used methods and their invocations per scenario. (1 – L4Linux with L4 console, 2 – L4Linux with L4 console and ORe, 3a –

Netfilter startup, 3b – Netfilter running, 4 – L4Linux with DOpE, 5a – Verner startup, 5b – Verner running, 6 – JTop, 7 – BLAC)

I sorted the functions by the message size to and from the server. I also highlighted the functions that make up at least 90% of the total IPCs in the respective scenario.

interface::method	1	2	3a	3b	4
2 words					
ore_notify::rx_notify	–	**17759**	**51533**	139	–
dm_generic::close	22	28	72	–	23
rmgr::init_ping	22	25	27	–	22
rmgr::get_task	316	316	347	161	316
rmgr::get_irq	16	16	16	–	16
rmgr::free_fpage	1	1	1	–	1
ts::free	14	14	27	80	14
ts::exit	–	–	–	–	–
verner::setPlayback	–	–	–	–	–
verner::setFxPlugin	–	–	–	–	–
overlay::input_listener	–	–	–	–	–
overlay::window_listener	–	–	–	–	–
overlay::create_window	–	–	–	–	–
overlay::destroy_window	–	–	–	–	–
overlay::open_window	–	–	–	–	–
overlay::set_background	–	–	–	–	–
3 words					
dm_generic::check_rights	–	–	–	–	1
dm_generic::transfer	46	55	89	3	46
dm_mem::size	16	12	14	–	23
dm_mem::resize	28	26	76	–	25
ts::allocate	19	19	45	85	19
ts::kill	28	28	48	160	28
ts::owner	1	1	3	–	1
con_vc::smode	1	1	3	–	–
con_vc::setfb	1	1	3	–	–
con_vc::direct_update	1953	2371	16235	**399103**	–
verner::setVolume	–	–	–	–	–
4–13 words					
dm_generic::share	6	7	21	–	6
dm_mem::physaddr	4	8	16	–	5
dm_phys::poolsize	73	179	615	2564	2
verner::changeQAP	–	–	–	–	–
verner::getPosition	–	–	–	–	–
overlay::open_screen	–	–	–	–	–
overlay::refresh_screen	–	–	–	–	–
dm_phys::pagesize	7	7	25	–	7
overlay::get_screen_info	–	–	–	–	–
overlay::stack_window	–	–	–	–	–
con_vc::gmode	1	1	3	–	–

continued on next page...

interface::method	1	2	3a	3b	4
verner::connect	–	–	–	–	–
overlay::place_window	–	–	–	–	–
con_vc::get_rgb	1	1	3	–	–
nitevent::event	–	–	–	–	–
names::register	9	10	10	–	13
names::query_name	62	111	126	18	70
rmgr::task_new	24	24	69	165	24
verner::start	–	–	–	–	–
names::register_thread	43	50	73	20	69
names::query_id	–	–	–	–	1
dm_mem::open	115	137	260	10	142
5 words and an indirect part of 20 words					
log::outstring	**7334**	**17765**	**58634**	**97394**	**6802**
258–267 words					
rmgr::get_task_id	10	11	12	–	10
rmgr::set_task_id	24	24	69	165	24
ts::create	33	33	69	165	33
dope::exec_cmd	–	–	–	–	**3152**
overlay::map_screen	–	–	–	–	–
dope::exec_req	–	–	–	–	1
515 words					
dope::init_app	–	–	–	–	1
266 words and an indirect part of 1KB size					
dopeapp::event	–	–	–	–	–
4–5 words and an indirect part of 4KB size					
ore_rxtx::send	–	**17755**	**51508**	494	–
ore_rxtx::recv	–	**17914**	**52039**	606	–
5 words including a flexpage					
rmgr::get_page0	1	1	1	1	–
6 words including a flexpage					
dm_generic::fault	**14578**	**14809**	**43900**	18	**14848**
7 words including a flexpage					
dm_generic::map	**12902**	**16747**	**35871**	8	**11434**

Table 6.12: Invocations for functions of scenarios (part 1)

interface::method	5a	5b	6	7
2 words				
ore_notify::rx_notify	–	–	–	**271344**
dm_generic::close	43	4	3	–
rmgr::init_ping	27	–	16	50
rmgr::get_task	64	–	64	734
rmgr::get_irq	16	–	8	27
rmgr::free_fpage	–	–	–	1
ts::free	–	–	–	178

continued on next page...

interface::method	5a	5b	6	7
ts::exit	–	–	–	1
verner::setPlayback	1	2	–	–
verner::setFxPlugin	1	4	–	–
overlay::input_listener	–	–	–	1
overlay::window_listener	–	–	–	1
overlay::create_window	–	–	–	17
overlay::destroy_window	–	–	–	7
overlay::open_window	–	–	–	18
overlay::set_background	–	–	–	1
3 words				
dm_generic::check_rights	–	102	2	–
dm_generic::transfer	379	477	44	–
dm_mem::size	121	**11413**	70	–
dm_mem::resize	–	179	–	–
ts::allocate	7	–	–	215
ts::kill	–	–	–	339
ts::owner	6	–	–	11
con_vc::smode	–	–	–	–
con_vc::setfb	–	–	–	–
con_vc::direct_update	–	–	–	–
verner::setVolume	1	5	–	–
4–13 words				
dm_generic::share	–	98	2	–
dm_mem::physaddr	184	–	–	–
dm_phys::poolsize	–	–	–	–
verner::changeQAP	1	1	–	–
verner::getPosition	1	428	–	–
overlay::open_screen	–	–	–	1
overlay::refresh_screen	–	–	–	1092
dm_phys::pagesize	**35643**	**3943**	34	–
overlay::get_screen_info	–	–	–	1
overlay::stack_window	–	–	–	78
con_vc::gmode	–	–	–	–
verner::connect	1	4	–	–
overlay::place_window	–	–	–	17
con_vc::get_rgb	–	–	–	–
nitevent::event	–	–	–	247
names::register	13	1	8	21
names::query_name	70	10	77	5275
rmgr::task_new	7	–	–	368
verner::start	1	8	–	–
names::register_thread	69	27	50	115
names::query_id	–	–	**1329**	25
dm_mem::open	1063	**11537**	156	–
5 words and an indirect part of 20 words				
log::outstring	2580	2056	**1767**	–

continued on next page...

interface::method	5a	5b	6	7
258–267 words				
rmgr::get_task_id	12	–	7	27
rmgr::set_task_id	7	–	–	370
ts::create	6	–	–	377
dope::exec_cmd	151	1189	**9668**	**116626**
overlay::map_screen	–	–	–	1
dope::exec_req	–	5	1	1210
515 words				
dope::init_app	1	3	2	2
266 words and an indirect part of 1KB size				
dopeapp::event	–	–	–	1197
4–5 words and an indirect part of 4KB size				
ore_rxtx::send	–	–	–	**284876**
ore_rxtx::recv	–	–	–	**288612**
5 words including a flexpage				
rmgr::get_page0	–	–	–	–
6 words including a flexpage				
dm_generic::fault	**48539**	**17249**	**1169**	–
7 words including a flexpage				
dm_generic::map	**66569**	**14694**	245	–

Table 6.13: Invocations for functions of scenarios (part 2)

As expected, the most messages in all startup scenarios are required to establish memory mappings in the address spaces of the applications. In all scenarios, the three most used functions make up at least 50% of the messages. In the scenarios "Netfilter running" and "Verner startup" almost 100%.

Döbel stated in [23] that 85% of all messages he observed in an configuration similar to the scenario "L4Linux with DOpE" were 24 words or less in size. The numbers shown in Tables 6.12 and 6.13 do not take messages into account that were not generated by an IDL compiler. For IDL compiler generated messages, about a third of all messages were 24 words or less in size. The scenario "Netfilter running" is an exception. Here all messages were 24 words or less in size.

For the scenarios "JTop" and "BLAC" I observed about 70% of the messages to be 256 words or larger in size. For the "JTop" scenario this has to be attributed to the heavy use of DOpE. For the "BLAC" scenario the numbers are distorted, because, as I mentioned above, I could not measure the invocations of the dataspace manager (mainly dm_generic::map and dm_generic::fault) and the logging service. These interfaces make up a great portion of the invocations.

In Section 2.4 I mentioned that Bershad et al. made the observation that 75% of all invocations go to 3 procedures and 95% to 10 procedures of the over 350 procedures present in their system. I made a similar observation with the exception of the scenarios "L4Linux with L4 console and ORe" and "Netfilter startup". In these two scenarios, the network package transport makes up one third of all IDL generated messages. Here, 75% of all invocations go to 5 procedures (instead of the original 3).

6.2 Performance Benchmark

I used Tables 6.12 and 6.13 to write an IDL file that contains all the methods involved in the above scenarios. The code generated from this IDL file has then been instrumented to measure its performance. This section will compare the stubs generated by DICE with stubs generated by other stub code generators or hand-written code. Further measurements are made on different architectures or with binaries generated by different compilers. All numbers are the average of 10,000 invocations of the same method. This will present results with warm caches.

Even though different methods have the same message sizes, the transmitted types vary. Different stub code generators generate different code for different types. Therefore, I kept the signatures (parameter list and types) of the original methods instead of combining same sized messages into one method.

6.2.1 Comparing Stub Code Generators for Fiasco

Table 6.14 shows the average round-trip time in cycles for the different methods. Measurements were performed on an Intel Celeron processor (Mendocino at 1595 MHz) with 512KB L2 Cache. All generated stubs were compiled using gcc version 3.4. The underlying microkernel is Fiasco. For brevity, I am only showing the ten functions that I identified as generating 90% of the IPCs. The full list of functions is shown in Table A.1.

Analysis of this data reveals that IDL^4 generated code is outperformed by all other stubs. This can be due to the fact, that IDL^4 originally targeted L4 version X.0. The support for L4 version 2 and L4 version X.2 (or version 4) has been added later and not much effort was spent on generating fast stubs. DICE generated code is faster than Flick generated code for most functions and especially for register IPC. For these functions its full optimization potential is recognizable. To my surprise, outperform DICE generated stubs dynrpc stubs, even though previous literature [35] suggests otherwise.

Table 6.14 contains only the functions that generate 90% or more of the overall messages. For the cells without performance numbers, the respective stub code generators failed to produce working stubs. As the graphical comparison in Figure 6.4 shows, DICE generated code outperforms code generated by other stub code generators. The two exceptions are `dope::exec_cmd` with an impact of about 100 cycles or 5% and `ore_rxtx::send` with about 700 cycles or 16%. Performance gains of DICE range from 3% (`log::outstring`) over 31% (`con_vc::direct_update`) to 64% (`ore_rxtx::recv`).

6.2.2 Comparing Stub Code Generators for Hazelnut

Table 6.15 shows the round-trip times in cycles for stubs generated by IDL^4 and DICE, compiled with gcc-2.95 and running on Hazelnut (L4 version X.0). The used processor was an Intel Pentium 4 (Willamette) with 256KB L2 Cache. I compared IDL^4 generated code, which is always inlined, with DICE generated code with non-inlined function calls and inlined function calls. The code generated by DICE with inlining performs better than code using function calls, but is still slower than IDL^4 generated code. This is due

interface::method	IDL[4]	Flick	dynrpc	DICE
ore_notify::rx_notify	4009	–	2383	1825
dm_mem::size	4778	3146	3790	2039
con_vc::direct_update	4163	3147	3037	2084
dm_phys::pagesize	5487	3890	3792	2043
log::outstring	4360	3205	3346	3236
dope::exec_cmd	4954	3260	3330	3466
ore_rxtx::send	6775	–	4283	4968
ore_rxtx::recv	52827	–	50069	17902
dm_generic::fault	–	7197	–	7093
dm_generic::map	–	7202	–	7081

Table 6.14: Performance of different stubs compiled with gcc-3.4

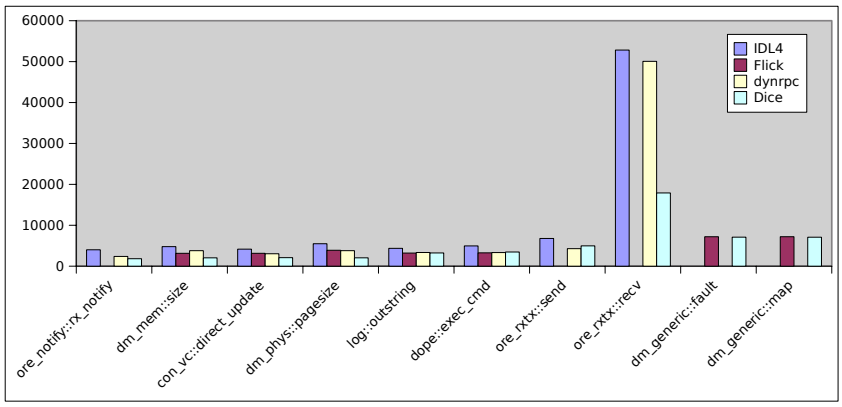

Figure 6.4: Comparison of selected generated stubs from different stub code generators running on Fiasco.

to the direct stack transfer optimization used in IDL^4 [51]. The gain due to optimization is about 300 cycles, which is 7%. This shows that architecture specific optimizations have a noticeable effect on performance.

interface::method	IDL^4	DICE	DICE with inlining
ore_notify::rx_notify	4091	4303	4175
dm_mem::size	4114	4423	4359
con_vc::direct_update	8889	4463	4378
dm_phys::pagesize	9033	9208	9160
log::outstring	9454	9672	9736
dope::exec_cmd	9650	10383	10184
ore_rxtx::send	19633	19913	19732
ore_rxtx::recv	–	14361	14468
dm_generic::fault	12308	12818	12679
dm_generic::map	17623	18020	17869

Table 6.15: Performance of different stubs running on L4 version X.0

L4 kernels that implement the version X.0 specification, transmit three words in a register IPC. This increases the number of functions able to invoke register IPC from 16 to 27, including two of the functions that make up 90% of the IPCs in one of the scenarios. A stub code generator that can exploit that knowledge can impact the performance of the overall system.

6.2.3 Comparing Hardware Architectures

Tables 6.16 and 6.17 compare the round-trip time of Dice generated code on different hardware architectures. The generated code was compiled using gcc-3.4.

The performance on the used AMD processors is comparable to the Intel Pentium 3. The Intel Pentium 4 processors perform much worse than any of the other architectures. This trend was reversed with Intel's more recent processors, Celeron and CoreDuo. As can be seen, the choice of the hardware platform has a large impact on the performance and can counterweight code optimizations performed by a stub code generator. But, as mentioned before, optimizing for the target architecture provides additional potential for performance improvements.

To fit the data onto these pages I split the table into one with Intel's processors and one with AMD's processors. As mentioned before, the AMD Opteron processor performs in the range of the Intel P3 processor. The AMD Athlon64 processor is comparable with Intel CoreDuo.

6.2.4 Comparing Compiler Versions

After comparing the generated stub code compiled with the same compiler running on different hardware platforms, I compared the performance of the generated stubs

interface::method	P1	P3	P4	Celeron	CoreDuo
rdstc	24	34	84	66	66
ore_notify::rx_notify	4084	2256	5214	1825	1867
dm_mem::size	5364	2629	5502	2039	2045
con_vc::direct_update	5477	2633	5403	2084	2059
dm_phys::pagesize	5221	2649	5541	2043	2059
log::outstring	9811	4160	7460	3236	3294
dope::exec_cmd	12035	4723	8059	3466	3468
ore_rxtx::send	19377	6947	9757	4968	4937
ore_rxtx::recv	41570	20573	50418	17902	17719
dm_generic::fault	21963	12227	15231	7093	7644
dm_generic::map	22162	12188	15382	7081	6694

Table 6.16: Performance of DICE stubs on different Intel processors

interface::method	Opteron	Turion	Athlon64
rdstc	10	10	9
ore_notify::rx_notify	2096	2099	1738
dm_mem::size	2490	2494	2071
con_vc::direct_update	2570	2571	2076
dm_phys::pagesize	2536	2542	2092
log::outstring	3434	3424	3507
dope::exec_cmd	3606	3613	3630
ore_rxtx::send	5057	4891	4939
ore_rxtx::recv	18012	18563	18024
dm_generic::fault	7509	8079	7007
dm_generic::map	7239	7960	6913

Table 6.17: Performance of DICE stubs on different AMD processors

compiled with different versions of the GCC running on the same hardware. The results are listed in Table 6.18.

interface::method	3.3	3.4	4.1	4.2	4.3
ore_notify::rx_notify	5254	5252	5184	5221	5081
dm_mem::size	5803	5551	5602	5572	5573
con_vc::direct_update	5698	5399	5598	5503	5612
dm_phys::pagesize	5802	5520	5627	5590	5621
log::outstring	7973	7491	7510	7498	7689
dope::exec_cmd	8466	8086	7977	7771	8075
ore_rxtx::send	10057	9721	9884	9920	9808
ore_rxtx::recv	50888	50505	165054	163490	166490
dm_generic::fault	15769	15372	15424	15859	15317
dm_generic::map	15830	15477	15410	15903	15397

Table 6.18: Performance of stubs compiled with different compiler versions

DICE generated stubs are small and optimized for the target architecture, which does not leave much room for compiler optimizations. A compiler needs some context to apply optimization strategies, which is not given for the small stubs.

To better visualize the improvements that different compiler versions can achieve I plotted a selection of the functions from above. Figure 6.5 shows the number of cycles of each function for different compiler versions in comparison with gcc-3.3. This figure points out the most prominent differences in performance for different compiler versions.

As can be seen, the compiler version has some effect on the performance of the generated stub. For some methods, such as `ore_notify::rx_notify`, new compiler versions have some limited benefit. For other methods, such as `verner::connect`, there is always an improvement with each new compiler version. There are also functions, such as `ts::exit`, where some versions of the compiler improved performance, and other versions worsened performance.

In contrast to Figure 6.5 I plotted the performance of the previously identified most used functions in Figure 6.6. Here you can see that the biggest impact is on the `ore_rxtx::recv` function. The usage of gcc-4.x brings threefold performance degradation. For the other functions, the effect of a newer compiler version is mostly positive, especially for function `dope::exec_cmd`, where gcc-4.2 compiled stubs are 8% faster than gcc-3.3 compiled stubs. The rest sees performance benefits of around 3%.

Using this data, one can say new compiler versions in general have a positive impact on the performance of generated stubs. Though, this impact is limited to a few percent for most of the functions, only.

6.2.5 Comparing Stub Code Generators for Pistachio

In this section I compare the performance of the stub code generated for the Pistachio microkernel by IDL[4] and DICE. The generated stubs were compiled with gcc-3.4 and

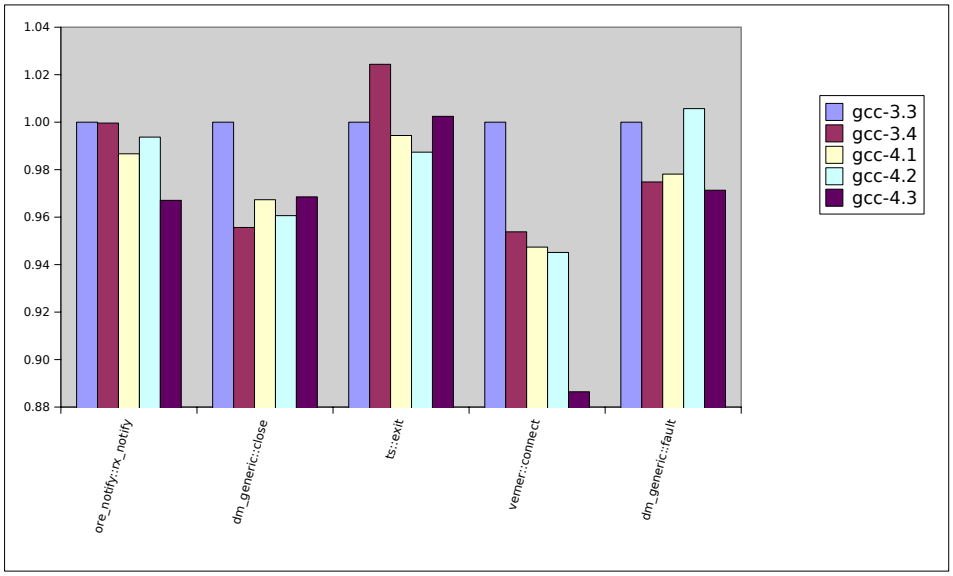

Figure 6.5: Performance of selected functions compiled with different compiler versions in comparison with version 3.3

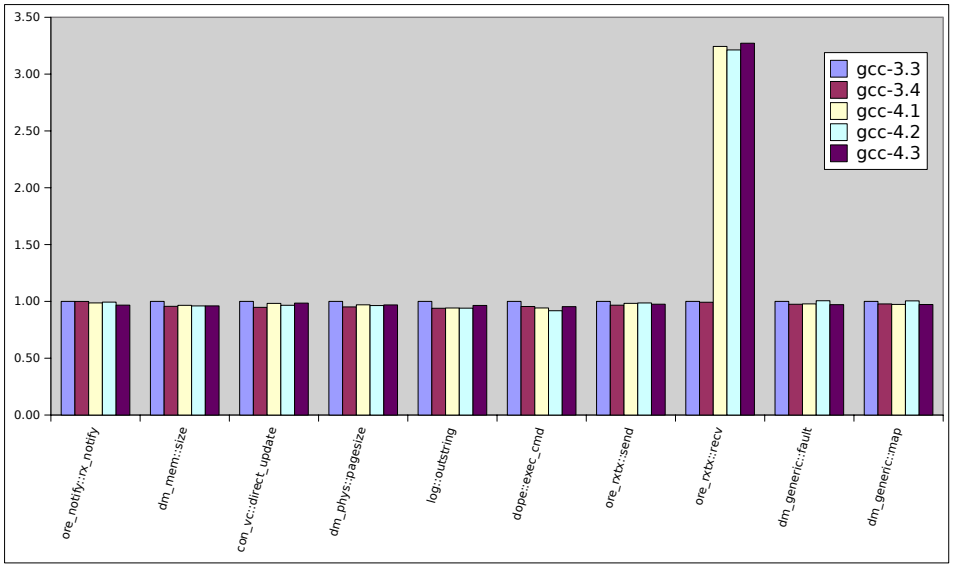

Figure 6.6: Performance of the most used functions compiled with different compiler versions in comparison with version 3.3

the binaries ran on a Pentium 4. The results are listed in Table 6.19.

interface::method	IDL^4	DICE
ore_notify::rx_notify	–	6287
dm_mem::size	2180	2609
con_vc::direct_update	2064	2531
dm_phys::pagesize	2173	2703
log::outstring	2018	8309
dope::exec_cmd	–	8472
ore_rxtx::send	–	17176
ore_rxtx::recv	–	43993
dm_generic::fault	–	5237
dm_generic::map	–	5409

Table 6.19: Performance of different stubs on Pistachio (L4 version X.2)

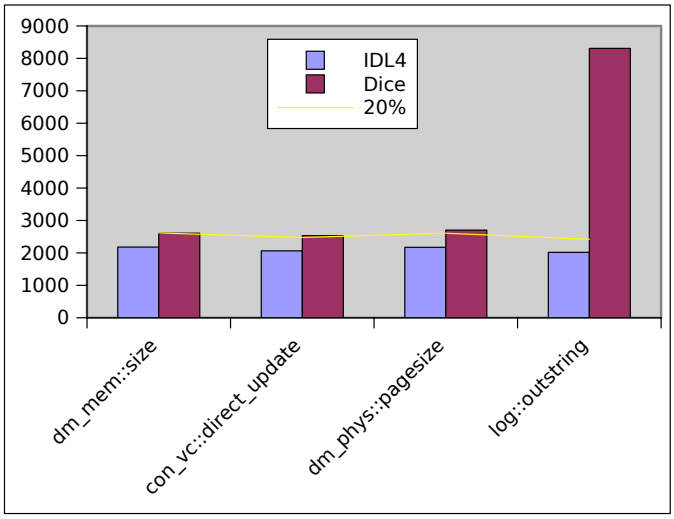

Figure 6.7: Performance of DICE generated stubs in comparison to IDL^4 generated stubs for most used functions on Pistachio.

Figure 6.7 only shows the most used functions, but almost all of the stubs generated by DICE are within 120% of the performance of the optimized stubs generated by IDL^4. There are a few instances, where DICE generated stubs performed better than IDL^4 generated stubs. There are several functions not depicted in this graph, for which one of the compiler could not generate a working stub.

Using indirect parts as mechanism for transport is implicit for stubs generated by IDL^4. The user has no possibility to specify if data should be transferred using indirect parts. For functions, where IDL^4 uses indirect parts and DICE doesn't, the IDL^4 generated stubs perform worse than the DICE generated ones. The one function, where DICE

performs worse than IDL^4 is the other way around: DICE generated code that uses indirect parts, because the IDL states so, but IDL^4 does not. IDL^4 can apply optimizations where DICE is bound to the contract of the IDL.

When comparing DICE to other stub code generators for L4, DICE generated stubs are almost twice as fast on its "native" platform–Fiasco. On "non-native" platforms the generated code is 7% to 20% slower. This demonstrates its ability to provide comparable stubs with its generic optimization framework.

6.2.6 Comparing Stub Code Generators for Linux sockets

The following section shows how I compare the performance of stubs generated for Linux socket interfaces. The generated code ran on a Pentium 3 and was compiled using gcc-4.2. This comparison shows that DICE generated stubs as well as dynrpc code perform worse than rpcgen generated code.

interface::method	rpcgen	dynrpc	DICE
ore_notify::rx_notify	39284	54105	31434
dm_mem::size	39913	44236	43585
con_vc::direct_update	40858	43022	41985
dm_phys::pagesize	40439	42822	45650
log::outstring	42232	43005	43017
dope::exec_cmd	41947	43170	54402

Table 6.20: Performance of different stubs for Linux sockets

In Figure 6.8, can be seen that DICE generated stubs and dynrpc stubs have a performance degradation compared to rpcgen stubs that lies within a 120% margin. An analysis of the stubs indicates that this difference might be due to the fact, that rpcgen client stubs are called with the same client object representing the connection to the server. The DICE generated stubs and the dynrpc stubs, on the other hand, created client sockets for every call. These two stubs also set the receive timeout before calling `recvfrom`. Because the Linux socket back-end in DICE and for dynrpc is for testing purposes only, no effort was put into tuning the stubs. This figure's performance numbers are percentages compared to rpcgen generated stubs, i.e., a stub with 105% is 5% slower than the rpcgen generated stub.

To investigate the difference of performance between the rpcgen generated stubs and the DICE generated and dynrpc stubs, I implemented the `default_timeout` functionality into DICE and dynrpc. The generated stubs do no longer set the receive timeout when waiting for a message, but use the socket's default timeout. The resulting comparison is shown in Figure 6.9. It shows that the performance of DICE and dynrpc generated stubs barely deviates from the rpcgen generated stubs with a few exceptions.

On average, the dynrpc stubs are faster than the DICE generated stubs. This may be contributed to a number of factors. Two are: different target languages of DICE and dynrpc. And: The DICE generated stubs contain elaborate error checking and handling.

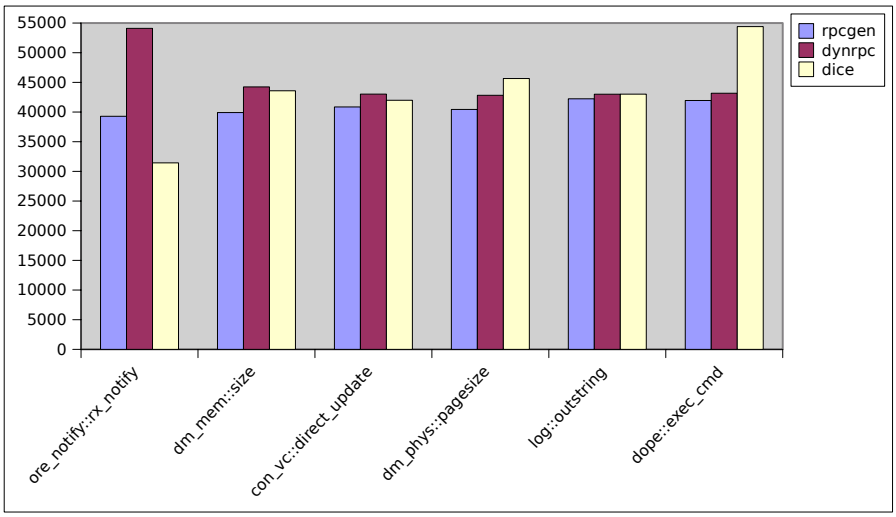

Figure 6.8: Performance comparison of DICE-generated and dynrpc stubs to rpgen-generated stubs

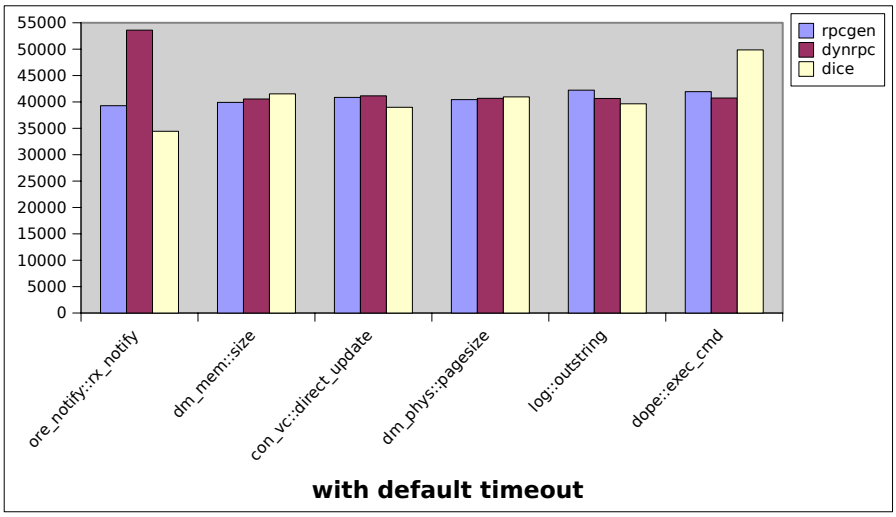

Figure 6.9: Performance comparison of DICE-generated and dynrpc stubs with default timeouts to rpgen-generated stubs

When building the dynrpc test, I encountered the limitation that for function specific features, such as using a default timeout for only one function, one has to either make that feature a run-time variable of the communication function or build a new communication class for this feature. The first approach affects all uses of this communication function by all callers. The second approach makes it hard to share state between functions of the same interface. In contrast, using attributes in the IDL file will generate code that is sensitive to the default timeout only for the functions with the attribute.

6.3 Micro-benchmarks

Additionally to measuring the performance of the stubs generated from the sample IDL file, I compare the performance of some special cases, such as register IPC and indirect string IPC versus direct IPC.

6.3.1 Short IPC

The fastest communication mechanism involving IPC on L4-based microkernel is register IPC, also called short IPC. Whenever a stub code generator generates code for short IPC, the generated stubs should add as little overhead as possible. I investigated different options of the stub code generation to determine the overhead of stubs generated by DICE and dynrpc.

I hand coded a scenario, where two threads in different processes exchange short IPC as fast as possible. Then I added simple dispatch functionality as would be found in a server. These numbers are compared with a DICE generated short IPC message stub. This setup includes the generated server loop. The first optimization feature I enabled was the usage of inlining. The generated client stubs as well as all generated server functions for which this is possible are inlined. Another feature I enabled was the use of default timeouts in the client stub. This saves accesses to the user-provided context (CORBA_Environment) containing the timeout value. This value is initialized to the default timeout and rarely set to a different value. Another optimization that I tested is the no exceptions attribute. For functions with this attribute, DICE does not generate exception test and unpack code.

I also measured the performance of dynrpc stubs. For these I wrote a simple client server call scenario, where the server replies immediately after receiving a message. Then I enabled the default timeout option for the dynrpc stubs and I implemented a basic dispatcher like the one I used for the hand written setup.

The DICE generated code for the inlined call stub with default timeouts introduces an overhead of 35% to the hand-coded dispatch code. Most of this overhead can be attributed to the fact that the used DICE version relies on the UTCB IPC feature of Fiasco. The DICE generated stubs at the server side pack and unpack messages from the UTCB. When a short IPC is sent, the two register values are stored in the UTCB by the generated code. This is done to unify the packing and unpacking of short IPC and longer messages.

scenario	cycles
hand-coded	1536
hand-coded dispatch	1512
dice call	2081
dice call inlined	2036
dice call inlined, default timeout	2042
dice call inlined, default timeout, no exceptions	2044
dynrpc call	2047
dynrpc call default timeout	2045
dynrpc call dispatched	2087
dynrpc call dispatched, default timeout	2088

Table 6.21: Performance of different short IPC scenarios.

Using the `no_exception` attribute does not yield any benefit, because the exception test and unpack code is already in a pessimistic code path. DICE generated stubs contain compiler hints as to which code path is the most likely one. This is used to minimize jumps on this path. Because exceptions are not likely to appear, this path is only taken when necessary. Eliminating this test and unpacking path does not gain much on the optimistic path.

The dynrpc test shows that a very short IPC call with a server similar to the hand-written one without dispatch, performs in the range of an inlined DICE generated call with full server and dispatcher. A dynrpc setup with a simplistic dispatcher performs in the range of an unoptimized DICE generated call.

These measurements were performed on a Fiasco kernel with debugging support enabled and frame pointer support. They are identified as performance critical. However, disabling these options would bereave the Fiasco kernel of functionality that is required to run real-world workloads, such as L4Linux. The overhead of 35% which is introduced by the generated stubs when compared to hand-written stubs is the same as the overhead identified in previous work.

6.3.2 Indirect string IPC versus direct IPC

Building on the previous identification of common message sizes in Section 6.1.8, I built a test case that measures the performance of these common message sizes. The key numbers to remember are 30, 300, 4KB indirect string and 4KB as flexpage. The 30 word sized message spans, what Döbel identified in [23] as 85% of all messages being 24 words or smaller. Also, the UTCB is 32 words in size, which allows for 30 word sized parameters.

A function transmitting a message of 300 words contains usually a buffer of size 1KB (256 words). I identified this type of message prevailing for some scenarios (JTop and BLAC). The 4KB indirect string messages are used by the network stack to transfer packages. And one page transfers are the means to establish shared memory.

For the transmission of different sized parameters I measured different setups:

- 30 variables on the stack, passed to the stub in sequential order
- 30 variables on the stack, passed to the stub in random order
- 30 global variables (in memory), passed to the stub in sequential order
- one array of 30 word sized elements
- one structure containing 30 word sized members
- one array of 60 word sized elements (to be larger than the UTCB)
- one array of 300 word sized elements

The measured round-trip times are listed in Table 6.22. As expected, there is a difference between arguments that are located on the stack to those arguments located in memory. In the various samples I also noticed that the sequential stack setup was faster than the random stack setup. On average, both setups use the same number of cycles.

scenario	cycles
30 stack, sequential	2134
30 stack, random	2133
30 memory, sequential	2159
30 array, stack	2568
30 struct, stack	2225
60 array, stack	4017
300 array, stack	4029

Table 6.22: Performance of different sized message transfer scenarios.

What surprised me a little is the fact that one array or a struct of size 30 words takes longer to pack and unpack than thirty single parameters.

To measure the performance of indirect string transfer, I transmitted indirect strings of various sizes. The results are shown in Table 6.23. Indirect strings are special in a way that the sender can specify an arbitrary source address and size to be transferred to the receiver. The receiver can specify any previously allocated buffer large enough to hold the message as receive buffer. This allows transferring data directly from a source location to a destination. With a UTCB IPC, data has to be copied from the stack or memory location into the UTCB and at the receiver's side out of the UTCB. To compare UTCB IPC to indirect string IPC, I measured the costs of copying data in memory (e.g., from memory into the UTCB or out of the UTCB).

When adding two times the cost of a copy operation to the time of a UTCB IPC, the UTCB IPC is faster than indirect string IPC as long as the message fits into the UTCB. As can be seen in Figure 6.10, when a message is larger than the UTCB, an indirect string IPC is faster.

data chunk size	memcpy	UTCB IPC + copy	indirect strings
16	50	2115	3092
32	50	2115	3118
64	57	2129	3130
128	73	2161	3169
256	105	2225	3231
512	153	2321	3266
1024	225	2465	3329
2048	369	2753	3482
4096	658	3331	3874
8192	1233	4481	4631
16384	2543	7101	7360
32768	10162	22339	13806

Table 6.23: Performance of memcpy operation for different sized memory chunks.

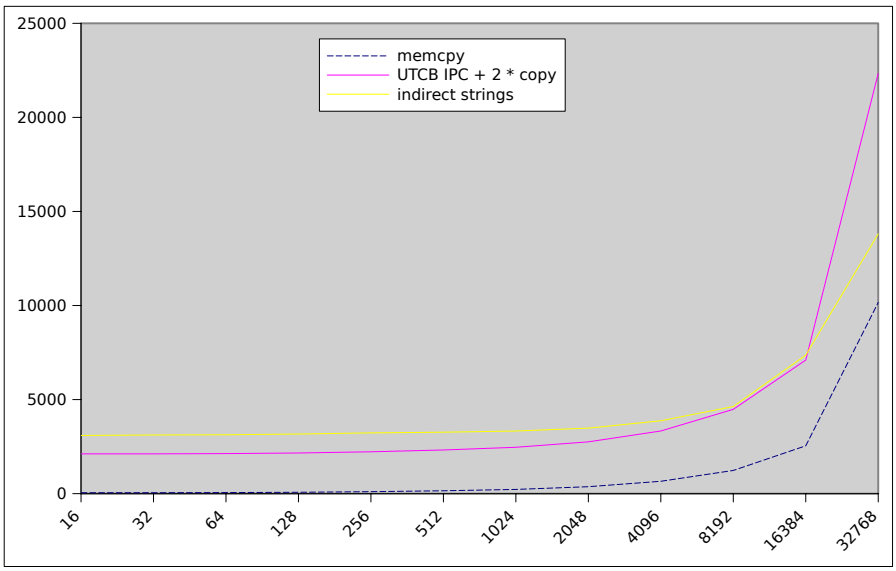

Figure 6.10: Performance comparison of indirect string IPC to UTCB IPC plus copy operations

6.3.3 Indirect string IPC versus Shared Memory

An alternative to indirect string IPC for large chunks of data is to establish a shared memory region between two address spaces. The cost for mapping one page is 10,143 cycles. To exchange data, it could be copied to and from that shared memory region. Notification about changes in this region could be done using shared locks or short notification IPCs. This setup would result in the onetime costs of setting up the shared memory region and the cost for copying data and notification for each transfer. I expressed these costs in Equation 6.1.

$$cost_{share} = cost_{map} + n\,(2cost_{copy} + cost_{notification}) \tag{6.1}$$

To find the number of invocations for which it is faster to use a shared memory region than indirect string IPC, we use the equation 6.2 as cost function.

$$cost_{indirect} = n\,cost_{indirect\_ipc} \tag{6.2}$$

The values of $cost_{map}$ and $cost_{notification}$ are known and constant at 10,143 cycles and 2,015 cycles respectively. The $cost_{copy}$ and $cost_{indirect_ipc}$ are functions depending on the size of the transmitted data. As long as $2 * cost_{copy} + cost_{notification} < cost_{indirect\_ipc}$ holds, than mapping will be faster. In Table 6.24 I compared the two values for the various message sizes. It shows, that up to a message size of 16KB it is faster to map memory, copy the data into this memory, and send a notification message than using indirect string IPC.

message size in bytes	copy plus notification	indirect string IPC
16	2115	3092
32	2115	3118
64	2129	3130
128	2161	3169
256	2225	3231
512	2321	3266
1024	2465	3329
2048	2753	3482
4096	3331	3874
8192	4481	4631
16384	7101	7360
32768	22339	13806

Table 6.24: Performance comparison of indirect string transfers to copy plus notification.

To avoid copy operations one could allocate data directly out of the shared memory. Also, the server can reference data instead of copying it out of the shared memory. At least the latter optimization is reasonable and would change Equation 6.1 to:

$$cost_{share} = cost_{map} + n\,(cost_{copy} + cost_{notification}) \tag{6.3}$$

To determine after how many message exchanges mapping actually is faster, we let $cost_{share} = cost_{indirect}$ and determine the n for different message sizes (see Equation 6.4.)

$$n = \frac{cost_{map}}{cost_{indirect\_ipc} - 2cost_{copy} - cost_{notification}} \tag{6.4}$$

For a message size from 16 to 512 bytes, n is 10. It means, after 11 message exchanges it is faster to use a shared region than indirect string IPC of the same size. For messages of 1KB at least 12 messages have to be exchanged, 14 messages for 2KB, 19 messages for 4KB and 136 messages for 8KB (here, two mappings have to be established). If fewer messages are exchanged, it is faster to use indirect string IPC.

6.4 Code Complexity

Different metrics exist when measuring code complexity. The simplest metric is counting physical lines of code for which I use *sloccount* [96]. Counting physical lines of code allows comparing projects written in different programming languages. Another metric is counting logical lines of code, which is highly language specific. The advantage of logical lines of code is different coding styles do not have an effect on the counted number of lines of code (LOC). One tool that I used to count logical lines of code is CodeCount [1] from the University of Southern California. Since I often compare projects written in different languages (IDL[4] and DICE in C++, Flick in C, Magpie in Perl) I will use CodeCount cautiously.

6.4.1 Test Suite

As mentioned in Section 5.2.4, DICE once integrated the generation of a test suite. The test suite could be used to call generated stubs and verify the correct transmission of parameters. Over time, the test-suite generation code became almost as complex as the stub code generation itself. To simplify DICE, I removed the test-suite generation and wrote a separate test suite instead. This new test suite allowed to integrate test for corner cases faster and could also repeatedly call stubs to allow easy memory leak detection.

The test-suite software could also be integrated into our automatic build environment. Using a user-level implementation of the Fiasco microkernel, I was able to test the execution of the test suite. Another approach to run the tests is the usage of a hardware emulator and to integrate its execution into the automatic build and test environment. This is an alternative if no user-level implementation of the microkernel exists.

When removing the test-suite generation code from DICE it was a little more than 4,300 LOC in size. Adding to that were special cases in parameter handling for the test suite. The parameter generating code had to know about shared variables, the different level of pointers to a variable, memory allocation for variables, etc. An array specified in the IDL as an integer pointer with a `size_is` attribute had to be translated into an

array that is allocated as global variable, shared between client and server and had to be appropriately initialized.

The replacement test suite I wrote, which tested the same functions was almost 3,900 LOC in size. Thus, I traded 4,300 LOC of test-generating code and additional complexity in DICE for a hand-written test suite of 3,900 LOC in size. The respective IDL file of the hand-written test suite is 277 LOC.

The current implementation of the hand-written test suite for DICE generated stubs grew to a little under 6,500 LOC without affecting DICE complexity. The IDL file for this test suite is 466 LOC big. Because setup and execution of the tests is similar between most of the functions, macros or templates could be used to further reduce the size of the test suite. Still, this approach has the freedom to add corner case tests, which require a specific setup or sequence of method invocations to trigger.

Generating communication stubs and testing communication stubs are two distinct problems. Trying to solve these two problems in one tool will make the tool overly complex. The selected solution to separate the stub code generation and the test suite proved that the IDL compiler becomes simpler.

6.4.2 Tracing

A previous implementation of the tracing framework, which included the generation of the complete tracing code in the IDL compiler, used 1516 LOC, or 3% of the total code base. Compared to other features this is a large piece of the compiler. Because of its size and inflexibility, I abandoned this integrated approach in favor of a plug-in architecture.

Today, the tracing framework in DICE consists of 200 LOC, or 0.4% of the code. This code sets up the tracing library, and invokes library callbacks at the hooks described in Section 5.2.3. There are 43 such hooks currently existing in DICE. A tracing library can be as little as 55 LOC or almost 800 LOC, depending on the objective. The smallest, currently implemented library adds one line to each client stub. The largest library adds 6 lines to each client stub and almost 20 lines to the server side code, or in other words, an average of 3 lines at every tracing hook.

6.4.3 Assembly code generation

In previous work [3] I have shown that generated assembler code yields an additional performance benefit of about 5% when compared to the respective C stub. The DICE stub code generator used the in-depth knowledge about the mechanism of the L4 IPC to achieve this performance benefit. Since then, the C/C++ compilers evolved and the advantage of using assembler stubs diminished. The C compilers were able to generate binaries from the generated C stubs that performed as good as the generated assembler code.

The portion of platform specific assembler code for the L4 version 2 API for IA32 architecture is 730 LOC or 1.5%. Table 6.25 contains a summary of the code size in different directories of the DICE stub code generator. Assembly code is generated by classes in the src/be/l4/v2/ia32 and src/be/l4/v2/amd64 directories.

directory	LOC C++	LOC yacc/lex	total
src	3622	–	3622
src/fe	5657	–	5657
src/parser	1145	–	1145
src/parser/idl	147	3572	3719
src/parser/c-c++	155	3560	3715
src/be	21764	–	21764
src/be/l4	3913	–	3913
src/be/l4/v2	1022	–	1022
src/be/l4/v2/ia32	730	–	730
src/be/l4/v2/amd64	123	–	123
src/be/l4/fiasco	1758	–	1758
src/be/l4/v4	1899	–	1899
src/be/sock	694	–	694
sum	42629	7132	49761

Table 6.25: Lines of code in subdirectories of DICE.

DICE provides most of its functionality in generic classes. The platform specific portions are comparatively small. This shows that the implementation of a new back-end involves little effort.

6.4.4 Indirect String Communication

To support indirect parts or indirect strings, two different sets of changes to the IDL compiler have to be considered. One is the invisible support as done in IDL[4], which includes knowledge about indirect parts, marshalling strategy, generating code to access indirect parts, etc. The other set of changes is the user-visible support, such as the possibility to specify indirect parts in the IDL and appearance of indirect parts in the generated function interfaces. The second set of changes relies on the first.

Indirect parts have to be supported internally (not visible to the user) to leverage the benefits of this mechanism. The visibility of this feature is arguable. I think it puts the optimization potential in the hands of the user. As always, allowing a user more freedom, also allows more freedom to misuse the feature.

	added to existing	new	total
front-end	0	11	11
back-end	83	501	584
sum			595

Table 6.26: Lines of Code to support indirect strings.

The numbers added to the front-end are solely additions to the parser and the in-memory representation of `ref` attributes and belong to the visible set. The changes

and additions to the back-end consist of both sets described above. Because the internal representation and back-end code generation cannot be separated completely, these numbers are combined. These changes combine to about 1% of the total code of DICE.

Giving the user the freedom to specify indirect parts in the IDL in exchange for 11 lines of code in the front-end and maybe 50 more in the back-end is, in my opinion, a reasonable tradeoff.

6.4.5 Resource Reservation

The support for resource reservation as introduced in Section 5.3.3 was implemented to inject calls to the resource reservation framework. The extension added support to read a resource description from the IDL file and generate the calls to the framework.

	added to existing	new	total
front-end	178	141	319
back-end	52	162	214
sum			533

Table 6.27: Lines of Code to support resource reservation.

These changes make up 2.2% of the front-end and 0.7% of the back-end. Overall, these changes make up 1.7% of the complete source code.

As shown, most of the code was used to parse the resource description in the IDL file. Putting the resource description next to the interface description allows associating resource usage and demand with specific – nameable – elements in the interface description.

As mentioned before, adding more functionality for less than 2% of the code base, or approx. 550 LOC, is a feasible approach. However, generated code heavily depends on the existence of the resource reservation framework. The generated stubs will not work or even compile without it.

6.4.6 Default Timeout

The default timeout support is an optimization feature. I have shown in Section 6.2.6 that this feature improves the performance of DICE generated and dynrpc stubs for Linux. For dynrpc stubs the performance gain was 5.5% on average and 8.2% for DICE generated stubs. In Section 6.3.1 I have shown the benefit for L4 stubs of 1.9%.

	added to existing	new	modified	total
front-end	11	0	5	16
back-end	3	132	41	176
sum				192

Table 6.28: Lines of Code of modification required to support default timeouts.

Given that support for this feature makes up 0.4% of the overall code, it shows that a wisely chosen optimization strategy can yield noticeable benefit.

6.4.7 Generated Code

For an analysis of the generated code I inspected the generated stubs of the scenario from Section 6.3.1. The generated call stub for a short IPC message is 62 LOC. The corresponding marshal and unmarshal functions at the server side are 63 LOC. The generic IPC exchange functions at the server side are 123 LOC. The generated dispatch function for one IDL method is 52 LOC and the generic server loop is 43 LOC. As one might suspect, the size of the call stub, the marshal and unmarshal function as well as the dispatch function depend on the number and type of arguments.

I used the generated stubs from the scenario described in Section 6.3.2 to determine the size of functions with more parameters. For the function that takes 30 distinct parameters a call stub of 147 LOC was generated. The corresponding marshal and unmarshal functions are also 147 LOC in size. These are almost 3 LOC per parameter: one in the call stub's parameter list, one in the message buffer type definition and one line for the actual marshalling.

For the function with one array parameter a call stub of 60 LOC was generated. This stub is smaller than the short IPC stub, because it receives no return value from the server. The corresponding marshal and unmarshal functions are also 60 LOC in size.

The server loop for the scenario from Section 6.3.2 is 45 LOC, which is two lines bigger than the server loop from the first scenario. The bigger server loop has to allocate a receive buffer for an indirect parameter. The generic IPC exchange functions are 129 LOC. The additional 6 LOC split in two times three additional lines to test for a received flexpage. This scenario's dispatch function is with 256 LOC bigger than the dispatch function of the short IPC scenario. This can be divided into 35 LOC of generic dispatch code plus code for each function. For a short IPC the function specific code is 18 LOC. For the function with 30 parameters it is 103 LOC. And the function with one array parameter uses 16 LOC. The required lines of code for a function in the dispatcher are 9 LOC of generic code plus the number of `[in]` parameters for the unmarshal function, the number of `[out]` parameters for the marshal function, and the number of all parameters for the actual function call.

I also measured the total number of lines of code of the scenario to determine the performance of all functions. The total LOC for Dice generated stubs is 21,906 LOC. This is an order of magnitude bigger than the hand-written code for dynrpc. However, these 22,000 LOC are highly tested and widely used code, whereas a dynrpc stub and server has to be written by hand and tested for each server individually.

I tried to compare the size of Dice generated stubs to the size of interpretive stubs as described in Section 2.5. The client side stub for a short IPC is almost 400 byte in size. The call stub for the 30 parameter function is almost 500 byte in size. The values are larger than the 300 bytes described in [63], so the benefit of using interpretive stubs should be even larger. However, such an approach trades performance against memory consumption.

6.4.8 IDL Compiler

Opposed to Flick, where each stage of the compiler is a separate binary, DICE integrates all stages into one binary. Flick's approach allows exchanging each single stage separately.

In DICE, different back-ends and front-ends are selected using run-time options. The downside of this approach is that the DICE binary has to be recompiled as a whole when a new front-end or back-end is added or modified. I expect this to be a rare event, and the associated overhead of recompilation acceptable.

The advantage of placing all stages into one binary is usability. It is easier to learn the usage of one tool, than three tools.

Front-End

The authors present in [29] numbers on the percentage of code reuse for each specialized target platform for the Flick IDL compiler. As mentioned above, Flick has three stages. The first stage–the front-end–corresponds to the DICE front-end, the second and third stage in Flick correspond to the DICE back-end. The lines of code for the front-end of Flick version 2.1 are shown in table 6.29. The percentage shown is the percentage of language specific code compared to the total lines of code of the front-end.

Component	Lines	
generic library	1973	
CORBA	2003	15.3%
ONC RPC	3009	23.0%
MIG	6098	46.6%
total	13083	

Table 6.29: Lines of Code for the Flick front-ends

The L4 specific changes to Flick for the front-end are almost negligible. A new type (flexpage) was added to the IDL.

Table 6.30 shows the lines of code for the DICE front-end. The total number of lines of code is about the same as for Flick's front-end. In DICE I combined the various IDL parsers into one parser and the C/C++ parsers into another. Each of the new parsers is about the average size of the language specific Flick front-ends. The generic library in DICE mostly consists of the classes for the in-memory representation of the parsed IDL file. The grammar files for the language make up a large portion of the parsers.

The IDL^4 compiler's front-end is smaller than DICE's front-end. Table 6.31 shows that the grammar for the IDL and C/C++ parsers is smaller than DICE's grammar.

Table 6.32 shows that the Magpie front-end is by far the largest of the IDL compiler's front-end. But the parsers are written in Python, which might explain their size compared to the generated parser of the other IDL compilers.

Component	Lines	
generic library	6802	47.8%
IDL parser	3719	26.1%
C/C++ parser	3715	26.1%
total	14236	

Table 6.30: Lines of Code for DICE front-ends

Component	Lines	
generic library	6498	56.2%
IDL	2267	19.6%
C/C++	2791	24.2%
total	11556	

Table 6.31: Lines of Code for IDL[4] front-ends

Component	Lines	
generic library	1912	6.0%
C parser	11817	37.2%
IDL parser	14831	46.7%
MIG parser	3218	10.1%
totals	31778	

Table 6.32: Lines of Code for MagPie front-ends

Back-End

The platform specific code generation is done by the DICE back-end. The majority of the back-end consists of basic infrastructure for all communication platforms and reflects the client-server architecture.

Currently DICE supports two major back-ends: L4 in different API versions and the Linux socket back-end. DICE used to support the Common Data Representation (CDR) format. Because of its missing acceptance, the CDR back-end became obsolete. The CDR back-end was 590 LOC in size plus 10 lines of code to recognize the respective command line option and generate the appropriate back-end class factory.

The CDR back-end is about the same size as the Linux socket back-end. Each makes up for around 2% of the DICE back-end. Thus one can say, that for just 600 LOC you can get support for a new communication platform in DICE.

In Table 6.33 I combined the numbers of Flick's presentation generator and back-end to put them side by side with the back-end of DICE.

Component	Lines	
generic library	15420	
CORBA IIOP	1562	2.1%
IIOP++	3584	4.8%
Mach 3/MIG	2253	3.0%
Khazana	1555	2.1%
Fluke	1065	1.4%
Trapeze	1578	2.1%
presentation conversion	47124	63.6%
total	74141	

Table 6.33: Lines of Code for Flick back-ends

Component	Lines	
generic library	21764	
Linux Socket	694	2.2%
L4 generic	3915	12.3%
L4 V2 API	1875	5.9%
L4.Fiasco API	1759	5.5%
L4 V4 API	1899	6.0%
total	31903	

Table 6.34: Lines of Code for DICE back-ends

Table 6.34 shows that the implementation of a back-end for a new communication platform in DICE requires adding between 2% and 6% to the existing code base, depending on the conformance with existing communication patterns.

When comparing Tables 6.29 and 6.30 and Tables 6.33 and 6.34 respectively, one can see that the front-ends of DICE and Flick are about the same size, as are the back-ends.

Component	Lines	
basic functionality	1394	
generic back-end	6468	
architecture generic	1174	
L4 V2 API	132	0.8%
L4 X0 API	4271	26.1%
L4 V4 API	2934	17.9%
total	16373	

Table 6.35: Lines of Code for IDL^4 back-ends

Component	Lines
generic back-end	2741
target architectures	995
templates	407
total	4143

Table 6.36: Lines of Code for MagPie back-ends

However, Flick adds the presentation conversion layer with another 47,000 lines of code. Thus, DICE can be considered moderate in size when compared to Flick.

Table 6.35 shows that the IDL^4 back-end is almost half the size of DICE's back-end. The smallest back-end of the compared IDL compilers has Magpie. As Table 6.36 shows, is it a little over 4,100 LOC in size.

The results of adding the size of front-end and back-end of the various compilers is shown in Table 6.37. When considering the features that each of the compilers supports, one can derive that more features and supported source language, as well as target languages, result in a bigger IDL compiler.

Compiler	total LOC
Flick	87224
DICE	46139
Magpie	35921
IDL^4	27929

Table 6.37: Total Lines of Code for different IDL compilers

6.4.9 dynrpc Code Complexity

A basic marshalling infrastructure for L4 version 2 IPC of the dynrpc framework is 357 lines of C++ code. I wrote a similar infrastructure for Linux network sockets in about an hour and it had a size of 445 lines of C++ code. This framework is almost negligible when compared to DICE.

I also compared the size of the accompanying client and server files for the test scenarios that I used to measure stub performance. These were 1772 lines of code for the client side and 2115 lines of code for the server. Additionally some 240 lines of code in common used header files were required, which totals the dynrpc test scenario to 4127 lines of code.

The message packing and unpacking has to be written by hand, which can lead to errors in the order of arguments in the message buffer. Also, the whole server loop and dispatcher has to be written by hand, which is a tedious task, since most server loops and dispatchers are the same. A server loop has 85 lines of code in average, the smallest (for one function) 38 lines of code, and the bigger ones adding approximately 13 lines for each additional function.

I compared this to the size of the test case for the DICE generated stubs. The client side code was 1272 lines of code; the server side required 1194 lines of code, no common header files but an IDL file of 363 lines of code. This sums up to 2829 lines of code, which is 1298 lines less than the dynrpc test, or 50% less to contain possible errors.

As I mentioned before, dynrpc can only be used with C++, whereas DICE supports C as well. Also, dynrpc relies on the C++ template libraries as trusted computing base.

6.5 Summary

In this chapter I showed that the optimizing IDL compiler I wrote performs better than non-optimizing and on par with other optimizing IDL compilers. I also showed, that its design is flexible and allows to add support for new features. This extensible design also allows implementing new optimization strategies, such as default timeout, easily.

With respect to code complexity, DICE is about the same size or smaller than other stub code generators for L4. It is bigger than alternative approaches, such as dynrpc, but also richer in functionality and flexibility.

Chapter 7

Conclusion and Outlook

This chapter provides a summary of my contributions and an outlook on possible extensions of this work.

7.1 Contributions

Communication in microkernel-based operating systems has many different forms. It is used to signal state, propagate rights, and for RPCs. Because components in a microkernel-based system often run in their own protection domains, inter-component communication involves crossing these protection boundaries and requires marshalling and unmarshalling of messages. With fast communication mechanisms to switch between protection domains, the stubs that setup, send, and receive messages make up a large portion of the communication. Stub code generators can be used to generate the communication code automatically from a small description of the communication interface. I addressed three concrete challenges for communication code generation in microkernel-based operating systems.

Forms of Communication

In Chapter 3 I showed that communication in microkernel-based systems is not only used to call components in other protection domains, but also to signal state changes, send one-way messages, or transfer rights to resources. When using a stub code generator, non-RPC forms of communication have to be supported. I discussed how each of the named forms of communication can be represented in an IDL and that the IDL compiler can generate well-performing code from it. This included the ability to use a generic communication framework, yet still exploit platform specific mechanisms.

In the Evaluation Chapter I demonstrated that the generated stubs also performed as good as or within close range of hand-written code and other stub code generators. These results show that a tool can be used to generate high-performance stubs.

Platform Independence

In Chapter 4 I explored how platform independence can be provided with an IDL but still platform specific optimizations can be exploited using attributes. I gave examples for using L4 specific communication mechanisms, such as indirect parts, or the transfer of flexpages. I also discussed optimization strategies for communication in generated stubs, such as parameter reordering or copy avoidance.

The platform independence allows using the same description of a component interface to generate communication code for different platforms. In the Evaluation Chapter I was able to give examples for this platform independence.

Feature Integration

Chapter 5 shows how existing designs for interface definition languages and their compilers can be integrated into an IDL compiler for microkernel-based operating systems. I discuss new extensions to the IDL to allow support of platform specific features, such as flexpages and indirect parts. I gave explicit examples for these different programming environments, such as resource accounting or resource reservation and how these can be supported in an stub code generator. I also discussed how real-time applications can use generated stubs and still maintain their timing guarantees.

Evaluation

In Chapter 6 I showed that the optimizing IDL compiler I wrote performs better than non-optimizing and on par with other optimizing IDL compilers. I also showed that its design is flexible and allows adding support for new features. This extensible design also allows implementing new optimization strategies, such as default timeout, easily.

With respect to code complexity, DICE is about the same size or smaller than other stub code generators for L4. It is bigger than alternative approaches, such as dynrpc, but also richer in functionality and flexibility and requires much less hand-written code.

7.2 Outlook

In the development of a trusted computing base, the tools used to translate part of the trusted components into binaries or generate communication code for the trusted components have to be trusted as well. To verify the correctness of the respective translation process, the tool – in our case the IDL compiler – has to be checked for correctness. A complete code size of about 50,000 lines of C++ code is hard to verify.

To ease the verification, the size of the compiler can be reduced. This can be done by removing features that can be implemented by wrapper libraries. Also, the use of more functionality from the standard libraries of the programming language can help to reduce the complexity of the compiler's code. A verification will then base on the correctness of the used standard library.

Methods to verify the correctness could be external tools, such as IDL compiler test tools [90].

Another approach to verify the TCB is to eliminate the use of the IDL compiler. Feske arguments in [35] along these lines. His approach, however, limits the use of dynamic RPC marshalling to one language, C++, and puts the burden of writing much of the boiler plate code of communication, such as marshalling stubs and server loops, back on the developer.

When I discussed access control in Section 3.5.1 I argued that access control is best implemented in the component functions. This discussion assumes that access control is based on a simple *subject, object, action* matrix. Another form of access control is to implement a stateful server. In the interface specification a valid sequence of function calls is defined with syntax similar to regular expressions. The IDL compiler computes from this valid sequence a state machine. Requests received by a server are matched to the state machine and allowed or rejected depending on the current state of the connection. Because a server should service multiple clients in parallel, it has to identify a connection to track the connection's state. Still, this approach does not hinder applications to use different interfaces provided by the same thread.

Appendix A

Performance Measurement Results

A.1 Performance of Stub-Code Generator for Fiasco

A.1.1 GCC version 3.4

The following number were measured on an Intel Celeron processor. The stubs have been compiled using gcc version 3.4 and were running on Fiasco.

interface::method	IDL[4]	Flick	dynrpc	DICE
2 words				
ore_notify::rx_notify	4009	–	2383	1825
dm_generic::close	2059	2430	3104	2058
rmgr::init_ping	3167	2463	2339	2058
rmgr::get_task	3746	2434	2340	2076
rmgr::get_irq	3188	2425	2327	2071
rmgr::free_fpage	2577	2469	2315	2074
ts::free	3758	3106	3086	2084
ts::exit	3749	2442	2353	2052
verner::setPlaybackMode	3761	2482	2618	2073
verner::setFxPlugin	3754	2434	2617	2075
overlay::input_listener	1989	3107	2334	2088
overlay::window_listener	4331	3131	2331	2108
overlay::create_window	2576	2425	2336	2081
overlay::destroy_window	3735	2427	2331	2098
overlay::open_window	3150	2429	2338	2106
overlay::set_background	3181	2400	2350	2081
3 words				
dm_generic::check_rights	4177	3170	3100	2080
dm_generic::transfer	4178	3143	3092	2079

continued on next page...

interface::method	IDL[4]	Flick	dynrpc	DICE
dm_mem::size	4778	3146	3790	2039
dm_mem::resize	3585	3169	3084	2081
ts::allocate	4170	3125	3053	2062
ts::kill	3550	3115	3074	2055
ts::owner	4153	3149	3057	2097
con_vc::smode	3008	3221	3043	2090
con_vc::setfb	4160	3177	3072	2087
con_vc::direct_update	4163	3147	3037	2084
verner::setVolume	4163	3162	3318	2061
4–13 words				
dm_generic::share	3618	3136	3105	2083
dm_mem::physaddr	6095	3890	3802	2065
dm_phys::poolsize	4821	3148	3099	2053
verner::changeQAP	5484	3882	3806	2103
verner::getPosition	4141	3139	5277	2073
overlay::open_screen	4170	3148	3120	2124
overlay::refresh_screen	4162	3128	3070	2075
dm_phys::pagesize	5487	3890	3792	2043
overlay::get_screen_info	4756	3149	3120	2125
overlay::stack_window	4142	3095	3069	2089
con_vc::gmode	4759	3302	3230	2216
verner::connect	4271	3893	3824	2150
overlay::place_window	4728	3103	3069	2121
con_vc::get_rgb	4811	3274	3120	2175
nitevent::event	5653	3093	3092	1899
names::register	3212	3155	3240	2155
names::query_name	6330	4540	3991	2744
rmgr::task_new	5452	3859	3860	2078
verner::start	4746	3167	3155	2083
names::register_thread	3785	3151	3243	2137
names::query_id	5416	3972	4298	2422
dm_mem::open	6222	4483	3944	2671
5 words and an indirect part of 20 words				
log::outstring	4360	3205	3346	3236
258–267 words				
rmgr::get_task_id	6197	3878	3977	4053
rmgr::set_task_id	3728	3163	3286	3375
ts::create	6254	3994	4068	4148
dope::exec_cmd	4954	3260	3330	3466
overlay::map_screen	5381	3351	3977	3614
dope::exec_req	6809	3980	4770	5318

continued on next page...

interface::method	IDL[4]	Flick	dynrpc	DICE
515 words				
dope::init_app	5003	3195	3428	4337
266 words and an indirect part of 1KB size				
dopeapp::event	4336	3238	3295	3525
4–5 words and an indirect part of 4KB size				
ore_rxtx::send	6775	–	4283	4968
ore_rxtx::recv	52827	–	50069	17902
6 words including a flexpage				
dm_generic::fault	–	7197	–	7093
7 words including a flexpage				
dm_generic::map	–	7202	–	7081

Table A.1: Performance of different stubs compiled with gcc-3.4

A.1.2 GCC version 2.95

The following number were measured on an Intel Pentium 4. The generated stubs were running on Fiasco and have been compiled with gcc version 2.95.

interface::method	IDL[4]	Flick	DICE
2 words			
ore_notify::rx_notify	10138	–	–
dm_generic::close	10902	6970	5512
rmgr::init_ping	8043	6912	5630
rmgr::get_task	19134	14055	13920
rmgr::get_irq	9495	7048	5512
rmgr::free_fpage	9435	7022	5463
ts::free	9530	10505	10428
ts::exit	9456	6949	5394
verner::setPlaybackMode	10356	7432	6436
verner::setFxPlugin	10327	7432	6333
overlay::input_listener	9585	10575	10391
overlay::window_listener	8124	10576	10393
overlay::create_window	8124	7124	5493
overlay::destroy_window	10997	7073	5442
overlay::open_window	8109	7017	5441
overlay::set_background	8168	7022	5435
3 words			
dm_generic::check_rights	18750	10597	10429
dm_generic::transfer	18718	10557	10431
dm_mem::size	15887	10654	5552

continued on next page...

interface::method	IDL[4]	Flick	DICE
dm_mem::resize	15877	10710	10454
ts::allocate	15917	10665	10412
ts::kill	17147	10500	10432
ts::owner	15816	10456	10372
con_vc::smode	17238	10427	10572
con_vc::setfb	17145	10555	10582
con_vc::direct_update	17238	10580	10605
verner::setVolume	18007	11051	11295
4–13 words			
dm_generic::share	17333	10563	10394
dm_mem::physaddr	24706	14126	13829
dm_phys::poolsize	17270	10842	10515
verner::changeQAP	21958	14084	14069
verner::getPosition	19225	15808	15761
overlay::open_screen	15820	10695	10358
overlay::refresh_screen	14298	10631	10413
dm_phys::pagesize	21902	14011	10657
overlay::get_screen_info	15968	10462	10573
overlay::stack_window	18802	10595	10321
con_vc::gmode	17265	10921	11025
verner::connect	26243	14106	14329
overlay::place_window	18620	10524	10389
con_vc::get_rgb	17184	10956	10916
nitevent::event	19281	10458	9706
names::register	23242	10687	11071
names::query_name	32724	15781	16060
rmgr::task_new	8015	6737	5507
verner::start	18686	10486	10676
names::register_thread	20374	10693	11040
names::query_id	26191	14359	14666
dm_mem::open	31088	15645	15757
5 words and an indirect part of 20 words			
log::outstring	25246	10597	13597
258–267 words			
rmgr::get_task_id	26644	14088	14275
rmgr::set_task_id	26100	10796	10870
ts::create	28102	14304	14802
dope::exec_cmd	24835	10743	11223
overlay::map_screen	24706	10960	11929
dope::exec_req	34052	14267	15622
515 words			

continued on next page...

interface::method	IDL[4]	Flick	DICE
dope::init_app	23708	10815	11251
266 words and an indirect part of 1KB size			
dopeapp::event	–	10828	13906
4–5 words and an indirect part of 4KB size			
ore_rxtx::send	28100	–	–
ore_rxtx::recv	38087	–	–
6 words including a flexpage			
dm_generic::fault	–	18949	18801
7 words including a flexpage			
dm_generic::map	–	18931	18933

Table A.2: Performance of different stubs compiled with gcc-2.95

A.2 Performance of Stub-Code Generator for Hazelnut

interface::method	IDL[4]	DICE	DICE with inlining
2 words			
ore_notify::rx_notify	4091	4303	4175
dm_generic::close	4110	4331	4374
rmgr::init_ping	4106	4459	4362
rmgr::get_task	4106	4443	4371
rmgr::get_irq	4102	4376	4451
rmgr::free_fpage	4106	4459	4367
ts::free	4102	4395	4358
ts::exit	4106	4370	4375
verner::setPlayback	4046	4286	4263
verner::setFxPlugin	4046	4379	4219
overlay::input_listener	4206	4568	4431
overlay::window_listener	4210	4572	4435
overlay::create_window	4210	4708	4620
overlay::destroy_window	4210	4548	4587
overlay::open_window	4206	4548	4503
overlay::set_background	4210	4551	4576
3 words			
dm_generic::check_rights	4106	4403	4279
dm_generic::transfer	4106	4287	4403
dm_mem::size	4114	4423	4359
dm_mem::resize	4106	4391	4371
ts::allocate	4110	4387	4258

continued on next page...

interface::method	IDL[4]	DICE	DICE with inlining
ts::kill	4110	4339	4379
ts::owner	4102	4368	4357
con_vc::smode	4086	4449	4399
con_vc::setfb	8885	4399	4359
con_vc::direct_update	8889	4463	4378
verner::setVolume	4044	4346	4307
4–13 words			
dm_generic::share	8895	9078	9080
dm_mem::physaddr	13873	14174	14030
dm_phys::poolsize	8934	9304	9160
verner::changeQAP	13837	14078	13986
verner::getPosition	13460	–	–
overlay::open_screen	9126	9460	9354
overlay::refresh_screen	9132	9493	9346
dm_phys::pagesize	9033	9208	9160
overlay::get_screen_info	9198	9438	9304
overlay::stack_window	9184	9511	9396
con_vc::gmode	9064	9258	9227
verner::connect	13885	14272	14103
overlay::place_window	9106	9465	9304
con_vc::get_rgb	8968	9268	9304
nitevent::event	9080	9160	9112
names::register	9450	9871	9741
names::query_name	10909	11394	11136
rmgr::task_new	9110	9369	9358
verner::start	8980	9158	9305
names::register_thread	9281	9692	9580
names::query_id	10840	10128	9950
dm_mem::open	15693	17292	17163
5 words and an indirect part of 20 words			
log::outstring	9454	9672	9736
258–267 words			
rmgr::get_task_id	9435	10354	10208
rmgr::set_task_id	9441	10278	10321
ts::create	9511	10241	10297
dope::exec_cmd	9650	10383	10184
overlay::map_screen	28562	11207	10921
dope::exec_req	138792	16464	16476
515 words			
dope::init_app	9623	11037	10997
266 words and an indirect part of 1KB size			

continued on next page...

interface::method	IDL[4]	DICE	DICE with inlining
dopeapp::event	9557	11016	11076
4–5 words and an indirect part of 4KB size			
ore_rxtx::send	19633	19913	19732
ore_rxtx::recv	–	14361	14468
5 words including a flexpage			
rmgr::get_page0	7433	12551	12727
6 words including a flexpage			
dm_generic::fault	12308	12818	12679
7 words including a flexpage			
dm_generic::map	17623	18020	17869

Table A.3: Performance of different stubs running on L4 version X.0

A.3 Comparing Hardware Architecture

interface::method	P1	P3	P4	Celeron	CoreDuo
rdstc	24	34	84	66	66
2 words					
ore_notify::rx_notify	4084	2256	5214	1825	1867
dm_generic::close	5348	2601	5483	2058	2054
rmgr::init_ping	5458	2581	5468	2058	2057
rmgr::get_task	5205	2613	5581	2076	2080
rmgr::get_irq	5196	2659	5526	2071	2058
rmgr::free_fpage	5623	2656	5381	2074	2072
ts::free	5250	2653	5770	2084	2082
ts::exit	5355	2696	5822	2052	2049
verner::setPlayback	5605	2668	5498	2073	2074
verner::setFxPlugin	5386	2662	5587	2075	2069
overlay::input_listener	5184	2635	5709	2088	2083
overlay::window_listener	5208	2694	5613	2108	2107
overlay::create_window	5406	2705	5649	2081	2072
overlay::destroy_window	5684	2754	5599	2098	2080
overlay::open_window	5088	2641	5625	2106	2079
overlay::set_background	5221	2669	5639	2081	2055
3 words					
dm_generic::check_rights	5249	2588	5431	2080	2070
dm_generic::transfer	5178	2589	5433	2079	2076
dm_mem::size	5364	2629	5502	2039	2045
dm_mem::resize	5317	2600	5439	2081	2066

continued on next page...

interface::method	P1	P3	P4	Celeron	CoreDuo
ts::allocate	5431	2655	5677	2062	2057
ts::kill	5244	2726	5677	2055	2054
ts::owner	5206	2714	5805	2097	2055
con_vc::smode	5307	2659	5504	2090	2090
con_vc::setfb	5364	2717	5530	2087	2071
con_vc::direct_update	5477	2633	5403	2084	2059
verner::setVolume	5491	2676	5539	2061	2062
4–13 words					
dm_generic::share	5437	2666	5508	2083	2046
dm_mem::physaddr	5505	2633	5567	2065	2059
dm_phys::poolsize	5251	2630	5546	2053	2080
verner::changeQAP	5727	2716	5610	2103	2096
verner::getPosition	5636	2749	5505	2073	2084
overlay::open_screen	5048	2692	5595	2124	2117
overlay::refresh_screen	5081	2648	5660	2075	2057
dm_phys::pagesize	5221	2649	5541	2043	2059
overlay::get_screen_info	5204	2766	5632	2125	2124
overlay::stack_window	5203	2670	5713	2089	2090
con_vc::gmode	5397	2853	5674	2216	2220
verner::connect	5709	2750	5568	2150	2107
overlay::place_window	5146	2672	5612	2121	2086
con_vc::get_rgb	5453	2723	5643	2175	2173
nitevent::event	4101	2311	5119	1899	1949
names::register	8466	2959	5892	2155	2297
names::query_name	6152	3292	7588	2744	2883
rmgr::task_new	5447	2693	5484	2078	2050
verner::start	5808	2741	5491	2083	2077
names::register_thread	5758	3019	5926	2137	2276
names::query_id	7683	3896	6974	2422	2736
dm_mem::open	14313	5867	10857	2671	4559
5 words and an indirect part of 20 words					
log::outstring	9811	4160	7460	3236	3294
258–267 words					
rmgr::get_task_id	13801	5562	9447	4053	3998
rmgr::set_task_id	11642	4734	8117	3375	3394
ts::create	14543	5803	9968	4148	4094
dope::exec_cmd	12035	4723	8059	3466	3468
overlay::map_screen	11547	5345	8655	3614	3624
dope::exec_req	15406	6588	10414	5318	4542
515 words					
dope::init_app	15013	5610	20061	4337	4507

continued on next page...

interface::method	P1	P3	P4	Celeron	CoreDuo
266 words and an indirect part of 1KB size					
dopeapp::event	12017	5322	8521	3525	3582
4–5 words and an indirect part of 4KB size					
ore_rxtx::send	19377	6947	9757	4968	4937
ore_rxtx::recv	41570	20573	50418	17902	17719
5 words including a flexpage					
rmgr::get_page0	19162	11201	14221	6436	6038
6 words including a flexpage					
dm_generic::fault	21963	12227	15231	7093	7644
7 words including a flexpage					
dm_generic::map	22162	12188	15382	7081	6694

Table A.4: Performance of DICE stubs on different Intel processors

interface::method	Opteron	Turion	Athlon64
rdstc	10	10	9
2 words			
ore_notify::rx_notify	2096	2099	1738
dm_generic::close	2472	2474	2056
rmgr::init_ping	2685	2513	2072
rmgr::get_task	2494	2495	2143
rmgr::get_irq	2491	2495	2063
rmgr::free_fpage	2496	2499	2063
ts::free	2496	2498	2063
ts::exit	2498	2501	2026
verner::setPlayback	2540	2543	2100
verner::setFxPlugin	2712	2540	2112
overlay::input_listener	2560	2568	2071
overlay::window_listener	2774	2598	2065
overlay::create_window	2539	2547	2068
overlay::destroy_window	2583	2589	2069
overlay::open_window	2585	2589	2077
overlay::set_background	2544	2548	2068
3 words			
dm_generic::check_rights	2476	2478	2061
dm_generic::transfer	2475	2481	2055
dm_mem::size	2490	2494	2071
dm_mem::resize	2482	2486	2059
ts::allocate	2518	2520	2031
ts::kill	2512	2515	2066

continued on next page...

interface::method	Opteron	Turion	Athlon64
ts::owner	2693	2522	2069
con_vc::smode	2607	2612	2078
con_vc::setfb	2585	2617	2077
con_vc::direct_update	2570	2571	2076
verner::setVolume	2568	2569	2101
4–13 words			
dm_generic::share	2495	2499	2059
dm_mem::physaddr	2517	2520	2076
dm_phys::poolsize	2502	2508	2077
verner::changeQAP	2613	2613	2177
verner::getPosition	2565	2569	2084
overlay::open_screen	2622	2629	2079
overlay::refresh_screen	2579	2585	2097
dm_phys::pagesize	2536	2542	2092
overlay::get_screen_info	2790	2619	2092
overlay::stack_window	2696	2635	2100
con_vc::gmode	2854	2687	2206
verner::connect	2663	2665	2213
overlay::place_window	2626	2633	2098
con_vc::get_rgb	2662	2667	2183
nitevent::event	2354	2352	1899
names::register	2764	2769	2247
names::query_name	3056	3063	2490
rmgr::task_new	2611	2614	2064
verner::start	2662	2663	2192
names::register_thread	2946	2774	2199
names::query_id	3389	3394	2737
dm_mem::open	4378	5178	4623
5 words and an indirect part of 20 words			
log::outstring	3434	3424	3507
258–267 words			
rmgr::get_task_id	4207	4266	4284
rmgr::set_task_id	3485	3493	3485
ts::create	4227	5173	4485
dope::exec_cmd	3606	3613	3630
overlay::map_screen	3608	3677	3738
dope::exec_req	4696	4789	4692
515 words			
dope::init_app	4252	4251	4289
266 words and an indirect part of 1KB size			
dopeapp::event	3808	3817	3835

continued on next page...

interface::method	Opteron	Turion	Athlon64
4–5 words and an indirect part of 4KB size			
ore_rxtx::send	5057	4891	4939
ore_rxtx::recv	18012	18563	18024
5 words including a flexpage			
rmgr::get_page0	6335	6349	6260
6 words including a flexpage			
dm_generic::fault	7509	8079	7007
7 words including a flexpage			
dm_generic::map	7239	7960	6913

Table A.5: Performance of DICE stubs on different AMD processors

A.4 Comparing Compiler Versions

interface::method	3.3	3.4	4.1	4.2	4.3
2 words					
ore_notify::rx_notify	5254	5252	5184	5221	5081
dm_generic::close	5817	5559	5627	5588	5634
rmgr::init_ping	5694	5487	5485	5462	5542
rmgr::get_task	5864	5596	5655	5547	5791
rmgr::get_irq	5629	5528	5472	5412	5643
rmgr::free_fpage	5657	5380	5535	5457	5581
ts::free	6011	5776	5791	5731	5780
ts::exit	5702	5841	5670	5630	5716
verner::setPlayback	5657	5480	5548	5387	5550
verner::setFxPlugin	5638	5478	5501	5449	5530
overlay::input_listener	5645	5680	5651	5623	5665
overlay::window_listener	5661	5611	5602	5683	5660
overlay::create_window	5635	5672	5620	5620	5666
overlay::destroy_window	5643	5595	5655	5587	5686
overlay::open_window	5677	5622	5691	5616	5659
overlay::set_background	5649	5620	5683	5572	5688
3 words					
dm_generic::check_rights	5726	5472	5577	5629	5574
dm_generic::transfer	5853	5480	5572	5514	5577
dm_mem::size	5803	5551	5602	5572	5573
dm_mem::resize	5911	5497	5645	5580	5580
ts::allocate	6058	5709	5641	5727	5726
ts::kill	5948	5677	5746	5652	5826

continued on next page...

interface::method	3.3	3.4	4.1	4.2	4.3
ts::owner	5920	5829	5794	5717	5805
con_vc::smode	5729	5479	5491	5497	5674
con_vc::setfb	5664	5532	5533	5459	5609
con_vc::direct_update	5698	5399	5598	5503	5612
verner::setVolume	5641	5494	5494	5430	5487
4–13 words					
dm_generic::share	5899	5538	5583	5575	5562
dm_mem::physaddr	5825	5517	5615	5590	5599
dm_phys::poolsize	5254	5570	5554	5617	5657
verner::changeQAP	5775	5544	5454	5414	5602
verner::getPosition	5613	5447	5371	5361	5559
overlay::open_screen	5810	5594	5605	5685	5704
overlay::refresh_screen	5786	5641	5646	5694	5653
dm_phys::pagesize	5802	5520	5627	5590	5621
overlay::get_screen_info	5832	5640	5717	5655	5686
overlay::stack_window	5768	5636	5699	5642	5708
con_vc::gmode	5899	5685	5594	5537	5647
verner::connect	5759	5493	5456	5443	5105
overlay::place_window	5766	5588	5771	5656	5668
con_vc::get_rgb	5786	5602	5469	5498	5666
nitevent::event	5188	5146	5038	5203	5525
names::register	6094	6083	6160	6015	5973
names::query_name	7686	7675	7518	7654	7584
rmgr::task_new	5653	5483	5489	5443	5575
verner::start	5685	5471	5683	5556	5536
names::register_thread	6135	5983	6023	6165	6112
names::query_id	7295	7057	7171	7135	7153
dm_mem::open	11413	11113	10857	10892	10928
5 words and an indirect part of 20 words					
log::outstring	7973	7491	7510	7498	7689
258–267 words					
rmgr::get_task_id	9966	9391	9199	9098	9682
rmgr::set_task_id	8700	8134	7713	7754	8422
ts::create	10060	9993	9974	9932	9829
dope::exec_cmd	8466	8086	7977	7771	8075
overlay::map_screen	8839	8636	8871	8999	8507
dope::exec_req	10632	10457	10453	10275	10071
515 words					
dope::init_app	14637	20054	8885	8736	9205
266 words and an indirect part of 1KB size					
dopeapp::event	8971	8514	8356	8161	8506

continued on next page...

interface::method	3.3	3.4	4.1	4.2	4.3
4–5 words and an indirect part of 4KB size					
ore_rxtx::send	10057	9721	9884	9920	9808
ore_rxtx::recv	50888	50505	165054	163490	166490
5 words including a flexpage					
rmgr::get_page0	14627	14289	14522	14504	14482
6 words including a flexpage					
dm_generic::fault	15769	15372	15424	15859	15317
7 words including a flexpage					
dm_generic::map	15830	15477	15410	15903	15397

Table A.6: Performance of stubs compiled with different compiler versions

A.5 Comparing Stub-Code Generators for Pistachio

interface::method	IDL[4]	DICE
2 words		
ore_notify::rx_notify	–	6287
dm_generic::close	2150	2567
rmgr::init_ping	2239	2541
rmgr::get_task	2357	2573
rmgr::get_irq	2347	2525
rmgr::free_fpage	2347	2551
ts::free	2171	2531
ts::exit	2035	2273
verner::setPlayback	2168	2559
verner::setFxPlugin	2197	2467
overlay::input_listener	2065	2435
overlay::window_listener	2065	2461
overlay::create_window	1992	2435
overlay::destroy_window	2059	2423
overlay::open_window	2058	2457
overlay::set_background	2059	2437
3 words		
dm_generic::check_rights	2177	2577
dm_generic::transfer	2187	2607
dm_mem::size	2180	2609
dm_mem::resize	2178	2655
ts::allocate	2161	2428
ts::kill	2166	2786

continued on next page...

interface::method	IDL[4]	DICE
ts::owner	2157	2597
con_vc::smode	2154	2611
con_vc::setfb	2069	2647
con_vc::direct_update	2064	2531
verner::setVolume	2218	2616
4–13 words		
dm_generic::share	2179	2627
dm_mem::physaddr	2229	2671
dm_phys::poolsize	2225	2613
verner::changeQAP	2263	2653
verner::getPosition	8097	5219
overlay::open_screen	2133	2606
overlay::refresh_screen	2046	2501
dm_phys::pagesize	2173	2703
overlay::get_screen_info	2105	2467
overlay::stack_window	2058	2495
con_vc::gmode	2150	2615
verner::connect	2210	2654
overlay::place_window	2045	2597
con_vc::get_rgb	2102	2672
nitevent::event	–	6550
names::register	7906	3036
names::query_name	8387	3230
rmgr::task_new	2307	2854
verner::start	2217	2579
names::register_thread	8002	2991
names::query_id	–	3555
dm_mem::open	8281	8781
5 words and an indirect part of 20 words		
log::outstring	2018	8309
258–267 words		
rmgr::get_task_id	8161	8425
rmgr::set_task_id	8319	8262
ts::create	8286	9099
dope::exec_cmd	–	8472
overlay::map_screen	–	–
dope::exec_req	–	–
515 words		
dope::init_app	–	13912
266 words and an indirect part of 1KB size		
dopeapp::event	–	13651

continued on next page...

interface::method	IDL[4]	DICE
4–5 words and an indirect part of 4KB size		
ore_rxtx::send	–	17176
ore_rxtx::recv	–	43993
5 words including a flexpage		
rmgr::get_page0	–	4962
6 words including a flexpage		
dm_generic::fault	–	5237
7 words including a flexpage		
dm_generic::map	–	5409

Table A.7: Performance of different stubs on Pistachio (L4 version X.2)

A.6 Comparing Stub-Code Generators for Linux sockets

interface::method	rpcgen	dynrpc	DICE
2 words			
ore_notify::rx_notify	39284	54105	31434
dm_generic::close	39625	42807	43720
rmgr::init_ping	39676	42710	43740
rmgr::get_task	40316	42587	46390
rmgr::get_irq	39858	42689	43810
rmgr::free_fpage	40000	42693	43736
ts::free	41670	42494	44252
ts::exit	39575	42641	44189
verner::setPlayback	39816	42619	41980
verner::setFxPlugin	39610	43068	41949
overlay::input_listener	39913	45855	44473
overlay::window_listener	39927	44561	44310
overlay::create_window	39498	44438	47013
overlay::destroy_window	39697	44350	44288
overlay::open_window	39503	44401	44281
overlay::set_background	41003	44460	44340
3 words			
dm_generic::check_rights	39857	42909	46340
dm_generic::transfer	40248	42746	45148
dm_mem::size	39913	44236	43585
dm_mem::resize	39943	42816	43731
ts::allocate	40230	43100	47119
ts::kill	40598	43260	44127

continued on next page...

interface::method	rpcgen	dynrpc	DICE
ts::owner	40327	44836	44322
con_vc::smode	40326	42944	41928
con_vc::setfb	40236	43381	44626
con_vc::direct_update	40858	43022	41985
verner::setVolume	40023	42688	41865
4–25 words			
dm_generic::share	40428	43054	43638
dm_mem::physaddr	42639	42869	43646
dm_phys::poolsize	40089	42723	43602
verner::changeQAP	40677	42741	41941
verner::getPosition	40020	43087	42395
overlay::open_screen	41628	44326	45208
overlay::refresh_screen	40264	44289	45766
dm_phys::pagesize	40439	42822	45650
overlay::get_screen_info	40275	44172	44390
overlay::stack_window	40329	44682	44427
con_vc::gmode	42131	42554	42062
verner::connect	41703	42905	46195
overlay::place_window	40399	44559	44372
con_vc::get_rgb	40553	42583	42082
nitevent::event	41120	42977	37369
names::register	42672	43423	43880
names::query_name	42084	44643	42471
rmgr::task_new	43114	43044	45319
verner::start	41628	44481	42183
names::register_thread	42145	43040	44877
names::query_id	45513	44155	43387
dm_generic::map	41350	42865	43841
dm_generic::fault	40586	42709	43732
dm_mem::open	42847	43228	47606
log::outstring	42232	43005	43017
258–267 words			
rmgr::get_task_id	42438	43009	48681
rmgr::set_task_id	42235	44448	51870
ts::create	44757	43082	49074
dope::exec_cmd	41947	43170	54402
overlay::map_screen	40278	44633	49651
dope::exec_req	42748	43780	56457
up to 1024 words			
dope::init_app	42633	44712	57443
dopeapp::event	42799	42888	60256

continued on next page...

interface::method	rpcgen	dynrpc	DICE

Table A.8: Performance of different stubs for Linux sockets

Bibliography

[1] Usc code count, 2006.

[2] M. J. Accetta, R. V Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for unix development. In *USENIX Summer Conference*, pages 93–113, Atlanta, GA, June 1986.

[3] Ronald Aigner. An idl compiler for microkernel systems, 2001. Master's thesis.

[4] Ronald Aigner. Integration of Message Passing into an IDL Compiler. In *2nd International Workshop on Microkernel-based Systems*, October 2001.

[5] Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. Towards Pervasive Treatment of Non-Functional Properties at Design and Run-Time. In *Proceedings of 16th International Conference "Software & Systems Engineering and their Applications (ICSSEA 2003)"*, Paris, France, December 2003.

[6] M. Aron, L. Deller, K. Elphinstone, T. Jaeger, J. Liedtke, and Y. Park. The SawMill Framework for Virtual Memory Diversity. In *Proceedings of the Sixth Australasian Computer Systems Architecture Conference (ACSAC2001)*, pages 3–10, Gold Coast, Australia, January 2001.

[7] Alexander B. Arulanthu, Michael Kircher, Carlos O'Ryan, Douglas C. Schmidt, and Anniruda Gokhale. Applying C++, Patterns, and Comonents to Develop an IDL Compiler for CORBA AMI Callbacks. *C++ Report magazine*, March 2000.

[8] Joshua S. Auerbach, Arthur P. Goldberg, German S. Goldszmidt, Ajei S. Gopal, Mark T. Kennedy, Josyula R. Rao, and James R. Russell. Concert/c: A language for distributed programming. In *USENIX Winter*, pages 79–96, 1994.

[9] Joshua S. Auerbach and James R. Russell. The Concert Signature Representation: IDL as intermediate language. *ACM SIGPLAN Notices*, 29(8):1–12, 1994.

[10] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004, LNCS*, volume 3362. Springer, 2004.

[11] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, October 2004.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of 22nd ACM Symposium on OS Principles (SOSP)*, Big Sky, MT, USA, October 2009.

[13] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already distributed. Why isn't your OS? In *Proceedings of 12th Workshop on Hot Topics in Operating Systems*, Monte Verita, Switzerland, May 2009.

[14] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight Remote Procedure Calls. In *ACM Transactions on Computer Systems*, pages 37–55, February 1990.

[15] Brian N. Bershad, Thomas E. Anderson, Edward D Lazowska, and Henry M. Levy. User-Level Interprocess Communication for Shared Memory Multiprocessors. In *ACM Transactions on Computer Systems*, pages 175–198, 1991.

[16] A. D. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[17] Matt Bishop. *Computer Security.* Addison-Wesley, 2003.

[18] David R. Cheriton and Willy Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP)*, pages 129–140, October 1983.

[19] COMQUAD - Components with Quantitative Properties and Adaptivity, 2002. http://www.comquad.org/.

[20] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 experimental kernel reference manual, version x.2. Technical report, University of Karlsruhe, 2004. Latest version available from: http://l4hq.org/docs/manuals/.

[21] Philip Derrin, Dhammika Elkaduwe, and Kevin Elphinstone. seL4 reference manual. Technical report, National ICT Australia, 2005. Latest version available from: http://www.ertos.nicta.com.au/research/sel4/.

[22] Philip Derrin, Dhammika Elkaduwe, and Kevin Elphinstone. *seL4 Reference Manual.* NICTA - National Information and Communications Technology Australia, 2006.

[23] Björn Döbel. Request tracking in DROPS, 2006. Diplomarbeit (Master's thesis).

[24] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. An Introduction to the MPI Standard, April 1995.

[25] Richard P. Draves. Everything You Always Wanted To Know About MIG. Slides to Rich Drave's talk on Mig, the Mach Interface Generator, November 1991.

[26] Richard P. Draves. Mach 3.0 IPC Internals. Slides to the talk by Rich Draves on 0ct 24, 1991 on the Internals of the Mach 3.0 Inter-Processes Communication system., October 1991.

[27] Richard P. Draves, Michael B. Jones, and Mary R. Thompson. MIG — The MACH Interface Generator. Unpublished manuscript from the School of Computer Science, Carnegie Mellon University., July 1989.

[28] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 189–202, Asheville, NC, December 1993.

[29] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–56, 1997.

[30] Eric Eide, Jay Lepreau, and James L. Simister. Flexible and optimized IDL compilation for distributed applications. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, volume 1511 of *Lecture Notes in Computer Science*, pages 288–302. Springer, May 1998.

[31] Eric Eide, James L. Simister, Tim Stack, and Jay Lepreau. Flexible idl compilation for complex communication patterns. *Scientific Programming*, 7(3-4):275–287, 1999.

[32] Kevin Elphinstone. Future direction in the evolution of the L4 microkernel. Technical Report 0401005T-1, National ICT Australia, October 2004.

[33] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. L4 reference manual—MIPS R4x00, version 1.0, kernel version 70. UNSW-CSE-TR 9709, University of New South Wales, School of Computer Science, 1998.

[34] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys 2006*, April 2006.

[35] Norman Feske. A Case Study on the Cost and Benefit of Dynamic RPC Marshalling for Low-Level System Components. *SIGOPS OSR Special Issue on Secure Small-Kernel Systems*, July 2007.

[36] Norman Feske and Christian Helmuth. Design of the Bastei OS Architecture. Technical Report TUD-FI06-07-Dezember-2006, TU Dresden, 2006. http://os.inf.tu-dresden.de/drops/doc.html#bastei-design.

[37] Nicholas FitzRoy-Dale. Magpie - interface compiler, 2004.

[38] Bryan Ford, Mike Hibler, and Jay Lepreau. Using Annotated Interface Definitions to Optimize RPC. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, page 232, December 1995.

[39] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and Execution Models in the Fluke Kernel. In *Proceedings of the 3rd USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 101–115, February 1999.

[40] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–151, October 1996.

[41] Open Software Foundation. DCE 1.1: Remote Procedure Call, August 1997.

[42] Torsten Frenzel. Design and Implementation of the L4.sec Microkernel for Shared-Memory Multiprocessors, 2006. Master's thesis.

[43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Sotware*. Addison-Wesley, 1995.

[44] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The Sawmill Multiserver Approach. In *ACM SIGOPS European Workshop*, September 2000.

[45] Derek G.Murray, Grzegorz Milos, and Steven Hand. Improving Xen Security through Disaggregation. In *Fourth Conference on Virtual Execution Environments (VEE)*, pages 151–160, Seattle, WA, USA, March 2008. ACM.

[46] Aniruddha Gokhale and Douglas C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *SIGCOMM Conference*, August 1996.

[47] Anniruda Gokhale and Douglas C. Schmidt. Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems. *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 17, September 1999.

[48] G.S. Graham and P.J. Denning. Protection—Principles and Practice. In *Spring Joint Computer Conference, AFIPS Conference Proceedings*, volume 40, pages 417–429, Montvale, N.J., 1972.

[49] Object Management Group. CORBA Messaging Specification, May 1998. OMG Document orbos/98-05-05 ed.

[50] The Open Group. *DCE 1.2.2 Introduction to OSF DCE*. The Open Group, November 1997.

[51] Andreas Haeberlen. Using platform-specific optimizations in stub-code generation, 2002. Study thesis.

[52] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-Code Performance is Becoming Important. In *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software*, San Diego, CA, October 22 2000.

[53] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[54] Christian Helmuth, Alexander Warg, and Norman Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.CH Security 2005*, Darmstadt, Germany, March 2005.

[55] Bernardo Hubermann, editor. *The Ecology of Computation*, Studies in Computer Science and Arti cal Intelligence. Elsevier Science Publishers, 1988.

[56] Galen Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. An Overview of the Singularity Project. MSR-TR 2005-135, Microsoft Research, October 2005.

[57] Joseph S. Barrera III. A Fast Mach Network IPC Implementation. In *USENIX MACH Symposium*, pages 1–12. USENIX, 1991.

[58] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using IPC redirection. In *7th Workshop on Hot Topics in Operating Systems (HotOS)*, Rio Rico, Arizona, March 1999.

[59] David B. Johnson and Willy Zwaenepoel. The Peregrine High-performance RPC System. *Software - Practice and Experience*, 23(2):201–221, 1993.

[60] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. In *OOPSLA*, pages 67–77, 1986.

[61] Bernhard Kauer. L4.sec Implementation Kernel Memory Management, 2005. Master's thesis.

[62] Bernhard Kauer and Marcus Völp. L4.sec kernel reference manual. Technical report, Technische Universität Dresden, October 2005. Latest version available from: `http://os.inf.tu-dresden.de/`.

[63] Peter B. Kessler. A Client-Side Stub Interpreter. *ACM SIGPLAN Notices*, 29(8):94–100, 1994.

[64] J. Liedtke. Lazy context switching algorithms for sparc-like processors. Arbeitspapiere der GMD No. 776, GMD — German National Research Center for Information Technology, Sankt Augustin, 1993.

[65] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[66] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[67] J. Liedtke. μ-kernels must and can be small. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 152–155, Seattle, WA, October 1996.

[68] J. Liedtke. Lava Nucles (LN) reference manual (486, Pentium, PPro) version 2.2. Technical report, IBM T.J. Watson Research Center, March 1998.

[69] J. Liedtke. L4 nucleus version x reference manual (x86). Technical report, University of Karlsruhe, September 1999.

[70] Jochen Liedtke, Nayeem Islam, and Trent Jaeger. Preventing denial-of-service attacks on a microkernel for weboses. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, MA, May 5–6 1997.

[71] Jork Löser, Lars Reuther, and Hermann Härtig. Position summary: A streaming interface for real-time interprocess communication. In *8th Workshop on Hot Topics in Operating Systems (HotOS)*, Elmau, Germany, May 2001. A comprehensive Tech Report is available: http://os.inf.tu-dresden.de/papers_ps/dsi_tech_report.pdf.

[72] Jork Löser, Lars Reuther, and Hermann Härtig. A streaming interface for real-time interprocess communication. Technical Report TUD-FI01-09-August-2001, TU Dresden, August 2001. Available from URL: http://os.inf.tu-dresden.de/papers_ps/dsi_tech_report.pdf.

[73] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, June 1995.

[74] Microsoft Interface Definition Language, 2007. http://msdn2.microsoft.com/en-us/library/aa367091.aspx.

[75] Sun Microsystems. *ONC+ Developer's Guide*. Sun Microsystems, 1995. Latest version available from: http://docs.sun.com/app/docs/doc/802-1997.

[76] Mark S. Miller and K. Eric Drexler. Comparative Ecology: A Computational Perspective. In Hubermann [55], pages 51–76.

[77] Mark S. Miller and K. Eric Drexler. Incentive Engineering: for Computational Resource Management. In Hubermann [55], pages 231–266.

[78] Mark S. Miller and K. Eric Drexler. Markets and Computation: Agoric Open Systems. In Hubermann [55], pages 133–176.

[79] Jim Mitchell. An Overview of the Spring System. In *IEEE Compcon*, February 1994.

[80] Markus Mock. Dce++: Distributing c++-objects using osf dce. In Alexander Schill, editor, *DCE Workshop*, volume 731 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 1993.

[81] B. J. Nelson. *Remote Procedure call*. PhD thesis, Xerox Palo Alto Research Center, Palo Alto, California, USA, 1981.

[82] John K. Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, 1988.

[83] Martin Pohlack, Ronald Aigner, and Hermann Härtig. Connecting Real-Time and Non-Real-Time Components. Technical Report TUD-FI04-01-Februar-2004, TU Dresden, 2004. Available from URL: http://os.inf.tu-dresden.de/papers_ps/tr-rtnonrtcomp.pdf.

[84] I. Pyarali, T.H. Harrison, and Dougas C. Schmidt. Asynchronous Completion Token: an Object Behavioural Pattern foe Efficient Asynchronous Event Handling. *Pattern Languages of Program Design*, 1997.

[85] Carsten Rietzschel. VERNER – ein Video EnkodeR uNd playER für DROPS, 2003. Master's thesis.

[86] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. In *Proceedings of the 12th ACM Symposium on Operating System Principles (SOSP)*, pages 83–90, December 1989.

[87] Jonathan S. Shapiro and Jonathan Adams. Design Evolution of the EROS Single-Level Store. In *2002 USENIX Annual Technical Conference*, Monterey, CA, USA, June 2002.

[88] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island Resort, SC, USA, December 1999.

[89] Lenin Singaravelu, Bernhard Kauer, Alexander Boettcher, Hermann Haertig, Calton Pu, Gueyoung Jung, and Carsten Weinhold. Enforcing Configurable Trust in Client-side Software Stacks by Splitting Information Flow. Technical report, GeorgiaTec University, November 2007.

[90] Pavel Strnad. IDL Source Generator and IDL Compiler Testing Tool, 2002. Master's thesis.

[91] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Libraries*. Springer-Verlag, 1997. International Standard ISO/IEC 8652:1995(E).

[92] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. Performance of the World's Fastest Distributed Operating System. *Operating Systems Review*, 22(4):25–34, 1988.

[93] C. van Schaik, B. Leslie, U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. NICTA L4-embedded kernel reference manual, version NICTA N1. Technical report, National ICT Australia, October 2005. Latest version available from: http://www.ertos.nicta.com.au/research/l4/.

[94] Linda R. Walmer and Mary R. Thompson. A Programmer's Guide to the Mach System Calls. School of Computer Science, Carnegie Mellon University., February 1988.

[95] Girish Welling and Maximilian Ott. Customizing idl mappings and orb protocols. In Joseph S. Sventek and Geoff Coulson, editors, *Middleware*, volume 1795 of *Lecture Notes in Computer Science*, pages 396–414. Springer, 2000.

[96] David A. Wheeler. More Than a Gigabuck: Estimating GNU/Linux's Size, July 2002. Version 1.07. Latest version available from: http://www.dwheeler.com/sloc.

[97] Michael Young, Avadis Tevanian, Richard F. Rashid, David B. Golub, Jeffrey L. Eppinger, Jonathan Chew, William J. Bolosky, David L. Black, and Robert V. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating System Principles (SOSP)*, pages 63–76, November 1987.